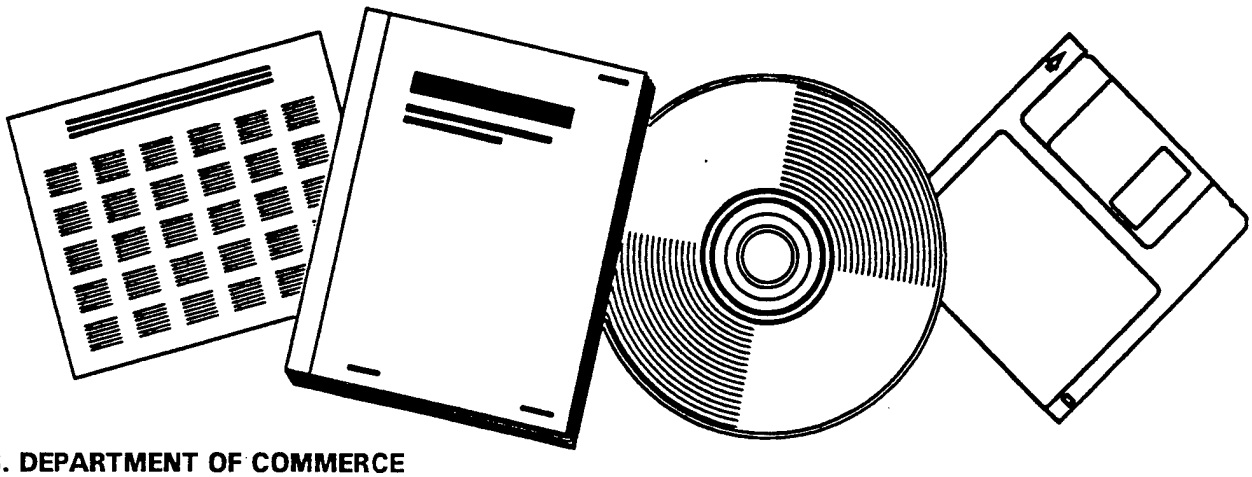
PB96-149802

DISTRIBUTING BACKWARD-CHAINING DEDUCTIONS TO MULTIPLE PROCESSORS

STANFORD UNIV., CA

APR 88



U.S. DEPARTMENT OF COMMERCE
National Technical Information Service

April 1988

Report No. STAN-CS-88-1224

Thesis

PB96-149802

Distributing Backward-Chaining Deductions to Multiple Processors

by

Vineet Singh

Department of Computer Science

Stanford University

Stanford, California 94305

DISTRIBUTING BACKWARD-CHAINING DEDUCTIONS TO MULTIPLE PROCESSORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Vineet Singh
April 1988

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188
Exp. Date: Jun 30, 1986

1a REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	Unlimited distribution
4 PERFORMING ORGANIZATION REPORT NUMBER(S) STAN-CS-88-1224	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Computer Science Department Stanford University 6b OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Stanford, CA 94305	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00039-83-C-0136 N00039-86-C-0033
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification)

Distributing Backward-Chaining Deductions to Multiple Processors

12. PERSONAL AUTHOR(S)
Vineet Singh

13a. TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
	FROM _____ TO _____	1988, April, 28	220

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

It is widely believed that parallel computation will be the basis for the next major advance in computing speed. In reality, many difficult problems remain to be solved. Two such problems are addressed in this thesis: (1) the design of a parallel execution model that exploits desirable types of parallelism; and (2) the design of a resource allocator to map the parallel computation to hardware resources for processing, storage, and communication.

The thesis presents a parallel execution model called PM for backward-chaining deduction with Horn clauses. For the target multiprocessor class, PM can exploit the most parallelism among existing execution models that use data-driven control. In particular, PM can exploit or-parallelism, and-parallelism, and pipelining.

The target class of multiprocessors has the following properties: (1) there are an arbitrary number of MIMD processors; (2) each processor has some local memory but there is no global memory; (3) processors can communicate only by sending messages to each other; (4) message delay is a function of the amount of data in the message and the distance between

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS		
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c OFFICE SYMBOL

19. Continued

source and destination; and (5) each processor can perform backward-chaining deductions based on the subset of the program that it contains.

The proposed allocation strategy is used at compile-time and applies to any application and multiprocessor (in the target multiprocessor class). However, it needs some restrictions that PM does not require. First, the type of backward-chaining deduction is restricted. In particular, no recursive clauses are allowed, unit clauses must be ground, and certain probabilistic uniformity and independence assumptions must apply. Second, a partitioning of the database is assumed to be given.

The allocator consists of a greedy allocation phase followed by a local minimization phase. Both greedy allocation and local minimization are based on a formally defined cost function that quantifies intuitive notions of undesirable allocations. Algorithms are presented for the efficient computation and recomputation of this cost function.

Considerable speedups are obtained by using this allocation strategy. These speedups compare favorably with an unreachable upper bound and speedups obtained using random allocations.

Abstract

It is widely believed that parallel computation will be the basis for the next major advance in computing speed. In reality, many difficult problems remain to be solved. Two such problems are addressed in this thesis: (1) the design of a parallel execution model that exploits desirable types of parallelism; and (2) the design of a resource allocator to map the parallel computation to hardware resources for processing, storage, and communication.

The thesis presents a parallel execution model called *PM* for backward-chaining deduction with *Horn clauses*. For the target multiprocessor class, *PM* can exploit the most parallelism among existing execution models that use data-driven control. In particular, *PM* can exploit *or-parallelism*, *and-parallelism*, and *pipelining*.

The target class of multiprocessors has the following properties: (1) there are an arbitrary number of MIMD processors; (2) each processor has some local memory but there is no global memory; (3) processors can communicate only by sending messages to each other; (4) message delay is a function of the amount of data in the message and the distance between source and destination; and (5) each processor can perform backward-chaining deductions based on the subset of the program that it contains.

The proposed allocation strategy is used at compile-time and applies to any application and multiprocessor (in the target multiprocessor class). However, it needs some restrictions that *PM* does not require. First, the type of backward-chaining deduction is restricted. In particular, no recursive clauses are allowed, unit clauses must be ground, and certain probabilistic uniformity and independence assumptions must apply. Second, a partitioning of the database is assumed to be

given.

The allocator consists of a *greedy* allocation phase followed by a local minimization phase. Both greedy allocation and local minimization are based on a formally defined cost function that quantifies intuitive notions of undesirable allocations. Algorithms are presented for the efficient computation and recomputation of this cost function.

Considerable speedups are obtained by using this allocation strategy. These speedups compare favorably with an unreachable upper bound and speedups obtained using random allocations.

Acknowledgements

I would like to thank Mike Genesereth (my principal adviser) for helping me become a better scientist. Interaction with members of Mike's Logic group (past and present) helped me a great deal in clarifying my ideas. I would like to single out Narinder Singh for his contributions.

I am grateful to the people at Fairchild Artificial Intelligence Research and later at Schlumberger Palo Alto Research for the use of their facilities, their financial support, and the interaction with some very talented individuals. Working with the FAIM multiprocessor group allowed me to test out my ideas in a more concrete setting. I would especially like to thank Al Davis and Bob Hon for their feedback.

I would also like to thank the rest of the members of my PhD committees—Ernst Mayr, Daniel Weise, and Thomas Kailath. Ernst Mayr, in particular, sparked many new ideas for my thesis as well as pre-thesis research.

My parents deserve credit for their early encouragement of my academic pursuits. I have always valued their support.

Finally, the biggest share of thanks must go to Kathy, my wife, for helping me have a wonderful time while I was working on my PhD.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

It is widely believed that parallel computation will be the basis for the next major advance in computing speed [45]. However, several difficult problems need to be solved. Two of these problems are addressed in this dissertation. The first problem is the design of execution models, or interpreters, that allow desirable types of parallelism to be exploited for certain types of computations. The second problem is the design of a resource allocator to map the parallel computation to hardware resources for processing, storage, and communication.

Optimal allocation is ruled out as a viable option because even simplistic computation and multiprocessor models make the problem NP-complete or worse [43]. A practical strategy must have the following characteristics: (1) hard limits on resources must be observed, (2) trade-offs must be made among the three types of hardware resources for processing, storage, and communication, and (3) the algorithms used for accomplishing resource allocation must themselves be reasonably efficient.

The type of computation being considered in this thesis is backward-chaining deduction [6]. This is the type of deduction employed in most extant logic programming languages. Prolog is a prime example.

Logical deduction is particularly attractive as a starting point for exploiting parallelism because (1) it has a well understood semantics that is completely independent of any computer architecture, be it sequential or parallel, and (2) it is not necessary for the programmer to be burdened with explicitly specifying the parallelism or for the interpreter/compiler to use complex techniques to uncover the parallelism. These two advantages together imply that the programmer can program *approximately* as he would with a sequential computer. We say *approximately* because optional *pragmas* (or hints) may be given by the programmer to increase the efficiency of parallel execution. This is analogous to the situation in which the programmer may do some explicit memory allocation/deallocation and leave most of the memory reclamation task to the garbage collector.

The rest of this chapter is organized as follows. Section 1.2 describes backward-chaining deduction. The next two sections describe the types of parallelism that can be exploited for this computation (section 1.3) and what is necessary to describe a *parallel execution model* (section 1.4). Section 1.5 describes the class of multiprocessors considered in this thesis and gives some background on FAIM-1, the specific multiprocessor that was used for experimentation. Section 1.6 gives the definition of the allocation problem and some background on previous allocation research. Section 1.7 describes the overall structure of the allocator that is described in detail later in the thesis. Finally, the last section presents the overall structure of the rest of the chapters in this thesis.

1.2 Backward-Chaining Deduction

Backward-chaining [6] is an inference mechanism for automated deduction. It is used here in the context of a database of *Horn* clauses. An example of a Horn clause is:

H :- T1,T2,...,Tn

where H and all Ti are positive literals (i.e., relation symbols with a list of terms). A term is a constant, a variable or a function of some terms. All variable names

will start with an uppercase letter. All constants, function symbols and predicate symbols will start with a lowercase letter. H is also called the *head* of the rule and the set T1,T2,...,Tn is called the *tail* of the rule. The meaning of the above Horn clause is:

H is true if all of T1,T2,...,and Tn are true.

By definition, if H is non-null, the clause is called an assertion (or fact) when n is zero and it is called a rule when n is greater than 0. If H is null, the clause is called a goal. For example,

$$:- G1,G2,...,Gm$$

is a goal with m literals. The meaning of this Horn clause is that the conjunction given below needs to be solved.

$$G1 \wedge G2 \wedge \ldots \wedge Gm$$

Solving a goal means proving it true or false (in the sense of logical implication). If the goal is true, then values of the variables in the goal that make it true must be given. The value of a variable is called a *binding*. A set of values for a set of variables is called a *substitution*.

An *and-or* tree is a problem reduction representation [7] used to represent the problem of proving a goal by backward-chaining. Figure 1 shows an example of a syntactic and-or tree used to represent a backward-chaining deduction. In this figure, ovals denote or-nodes and boxes denote and-nodes. And-nodes get their name because the goal they represent is one conjunct in a conjunctive goal set. Similarly, or-nodes represent a disjunct in a disjunctive goal set. Nilsson [48] gives a more formal characterization of and-or trees. Arcs are marked with the number of the clause used for the reduction. Also, a cut through the arcs going from a node to its children indicates that the children are and-nodes. The leaf nodes cannot be reduced. Leaf nodes may be empty boxes. These denote empty goals (i.e., successes). All leaf nodes in the example are empty goals. In other cases, a non-empty leaf indicates a failure. A *logical inference* is defined as the reduction of a goal

by a rule. In this example, substitutions that make the goal true are {X=a,Y=b} and {X=b,Y=a}. The discerning reader will notice that the former substitution can be obtained in two different ways (of proving the top level goal).

We call the tree syntactic because certain subtrees may be instantiated multiple times during an actual execution. For example, if conjuncts are solved left to right, multiple solutions to "p(X)" will lead to as many instantiations of the subtree rooted at "q(Y)".

Some of the leaf nodes in the and-or tree end in failure and others end in success. The purpose of the backward-chaining inference procedure is to find either one or all nodes associated with success in the and-or tree. Each node represents a solution. Therefore, the computation is a search problem. In this thesis, we restrict our attention to the case in which all solutions are desired.

The most widely used sequential interpretation is the one used by Prolog. The search through the tree is a depth-first, left-to-right search. Search is suspended for solutions to a subgoal when one solution is found. Search continues for the next solution by chronological backtracking from the next conjunct. When the first answer is obtained to the top level goal, it is announced. If more solutions are demanded, the search continues. Parallel approaches to interpretation are discussed in the next chapter. In particular, a parallel execution model called *PM* is described. *PM* exploits more types of parallelism than other execution models that use data-driven control and non-shared memory multiprocessor architectures.

The computation studied in this thesis is very similar to Prolog but not identical. In particular, a couple of features that are part of Prolog are not allowed here. First, Prolog programs can change the database of horn clauses. Side-effects of this type are not allowed in this thesis. Second, Prolog programs allow "cuts"—a construct used to prune part of the search space. "Cuts" are not allowed in this thesis. The allocation strategy imposes additional restrictions on the computation as will be seen later.

Top level goal		r(X,Y)
Database	C1	r(X,Y):-p(X),q(Y),s(X,Y).
	C2	r(X,Y):-f(X,Y),g(X),h(Y).
	C3	p(a).
	C4	p(b).
	C5	q(Y):-m(X),n(X,Y).
	C6	m(a).
	C7	m(b).
	C8	n(b,a).
	C9	n(b,b).
	C10	s(a,b).
	C11	s(b,a).
	C12	f(a,b).
	C13	g(a).
	C14	g(b).
	C15	h(a).
	C16	h(b).

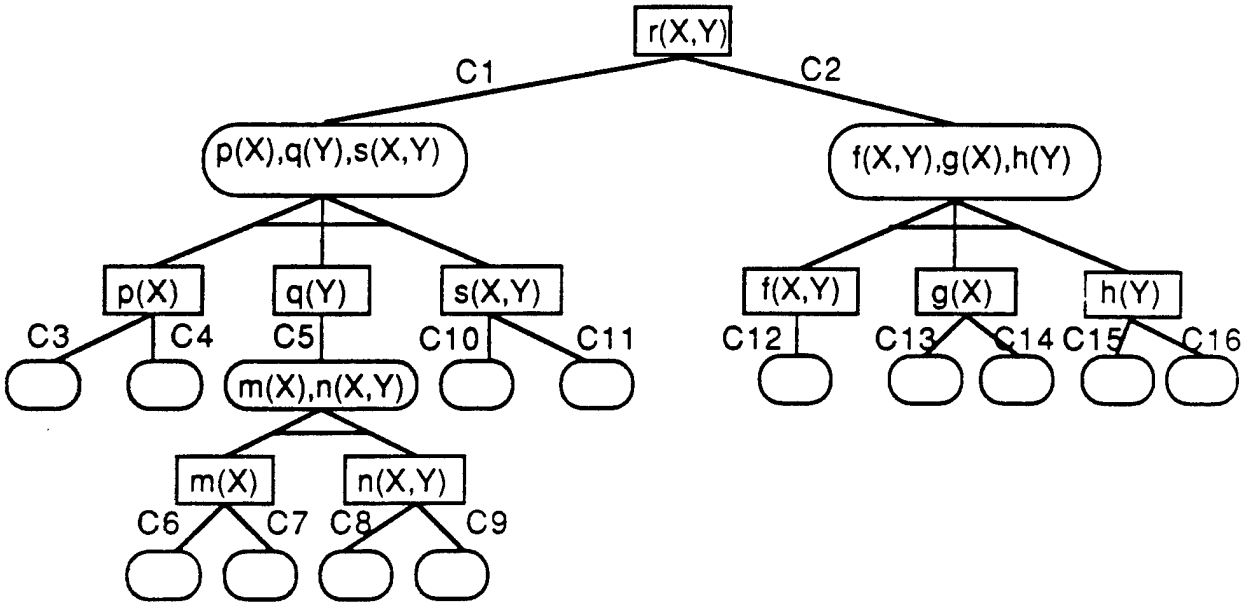


Figure 1: A Syntactic And-Or Tree

1.3 Types of Parallelism

Several types of parallelism have been described in the literature. The list below may not be exhaustive but covers the well-known types.

1. Or-parallelism: This is the solution of multiple or-nodes in parallel. There is some disagreement in the literature about the exact meaning of this. The most commonly used meaning [41], and the one used in this thesis, is that the entire search trees rooted at the or-nodes can be searched in parallel. In figure 1, the two sub-trees rooted at the two children or-nodes of the and-node "r(X,Y)" can be searched in parallel using or-parallelism.

   Conery [17,16] uses a slightly different meaning of or-parallelism. He defines or-parallelism as the assignment of a process to each or-node. Presumably, this meaning is neutral about the parallel search of the rest of the sub-trees below the or-nodes.

2. And-parallelism: This is the parallel solution of sibling and-nodes. Note that this does not mean that the and-nodes must be solved in isolation from each other or that they must all be solved in parallel. In figure 1, the and-nodes "p(X)" and "q(Y)" may be solved in parallel using and-parallelism.

3. Pipelining: This is the continuous streaming of complete solutions from one and-node to another. This is useful when two and-nodes must be solved in sequence. For example, pipelining allows the first solution of a source and-node to be sent to a destination and-node and allows the parallel search for (1) the first solution of the destination and-node and (2) the second solution of the source and-node. In figure 1, it may be the case that the and-nodes "m(X)" and "n(X,Y)" are solved in sequence. Using pipelining, solutions of "m(X)" can be streamed continuously to "n(X,Y)". The search for consistent solutions for "n(X,Y)" can begin as soon as a solution of "m(X)" is received.

4. Search-parallelism: This is the parallel reduction of an and-node to its children or-nodes. The term "search" refers to the search for clauses whose heads unify

with the and-node. The actual solution of the or-nodes in parallel is called or-parallelism (as defined above). In figure 1, the literal "r(X,Y)" can be unified with the heads of the two relevant rules in parallel.

5. Stream-parallelism: Conery [17] defines this as the "eager evaluation of structured data, which can be treated as a stream". Conery cites the example of testing for membership in a list while the list is still being constructed. There is no example of this in figure 1 and this type of parallelism is not considered in this thesis. Examples of this can be found in the work of Shapiro [57] among others.

6. Unification-parallelism: This is the parallelism associated with the unification of two literals. It has been shown that this problem is inherently non-parallelizable [20,74] (since it falls outside the problem class NC unless NC = FP). In attempting to exploit unification-parallelism, the hope is that practical cases of unification-parallelism are more amenable to speedup. Again, this type of parallelism is not considered in this thesis. Examples of this can be found in the work of Citrin [13] and Robinson [53] among others.

1.4 Parallel Execution Models

A Parallel Execution Model for a sequential program and a multiprocessor contains the specification of (1) methods to generate a set of parallel processes, (2) the state, procedures, and inter-process communication for the set of processes, and (3) any constraints placed on how the set of processes must be run on the processors in the multiprocessor.

The Parallel Execution Model is correct iff it produces the same solutions as the sequential program.

Same can mean the same set of solutions or the same ordered set of solutions. In this thesis, we use the former meaning (i.e., the order in which the solutions are produced is not considered significant).

For example, the set of parallel processes shown in figure 2 might be able to perform the backward-chaining associated with the and-or tree shown in figure 1. The arrows in the figure show communication of data or control. As the figure also shows, the state, procedures, and messages associated with process "s(X,Y)", as well as all other processes, must be specified. In our case, the parallel execution model is said to be correct iff the set of solutions produced by it is equal to the set of solutions produced by the Prolog interpreter as described in section 1.2.

A parallel execution model needs to exploit as much parallelism as possible while not being too complicated or expensive (in time and space) to be practical. These two requirements are clearly inconsistent, in general, and a reasonable tradeoff must be made.

A dataflow representation of the computation is desirable for exploiting concurrency. There are at least two important reasons. First, a dataflow representation of a computation makes all its parallelism explicit. Second, it has been argued convincingly that reasoning about dataflow programs for purposes of proving correctness properties and allocation is easier than reasoning about other procedural representations [5,11].

Although, a dataflow representation is desirable, it is not so at any cost. For example, Fortran programs may be reformulated as dataflow programs but at the cost of extensive copying of structures. The same argument holds for any other procedural representation that allows modification of global state. Fortunately, for logic programs, it has been shown that they can be represented easily as dataflow programs (with indeterminate merge) if the types of parallelism to be exploited are *or-parallelism* and *pipelining* only (see work by Ciepielewski and Haridi [12], Lindstrom and Panangaden [41], and Singh and Genesereth [61]). Conery [15] has shown how to exploit *or-parallelism* and a restricted form of *and-parallelism*, but not *pipelining*. However, the control mechanism was not data-driven in nature, but was a variant on the sequential backtracking mechanism of Prolog. *PM*, the parallel execution model presented in this thesis, shows how to exploit all three types of parallelism, *or-parallelism*, *pipelining*, and the same restricted form of *and-parallelism*

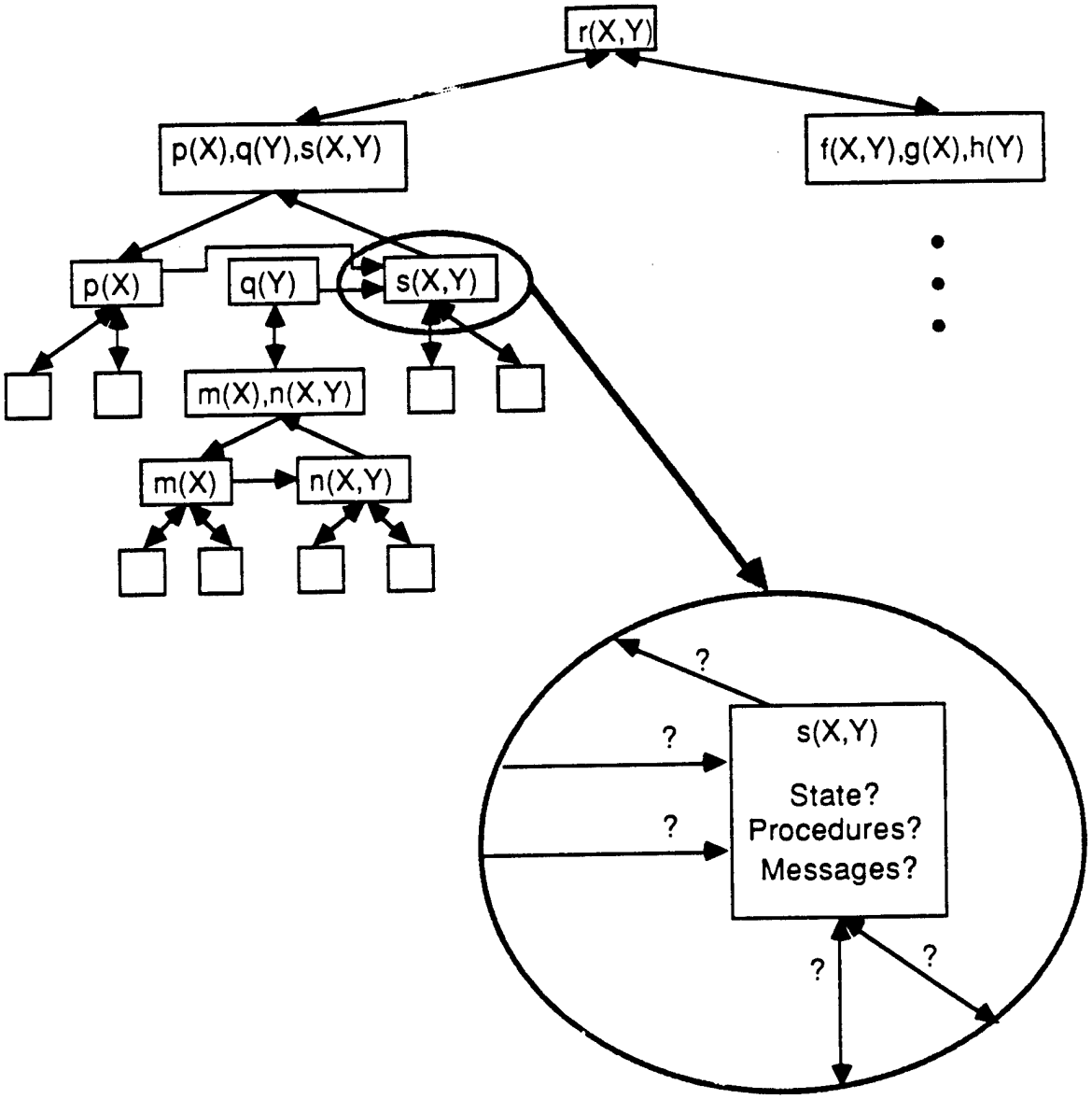


Figure 2: A Parallel Execution Model

described by Conery, while still using a data-driven solution. However, one more extension, local state, had to be made to dataflow (other than indeterminate merge). Local state makes the programs harder to reason about but the hope is that the reasoning is still far easier than it is for arbitrary procedural representations with global state (like Fortran). The resource allocation algorithms described in this thesis illustrate this ease of reasoning to some extent.

On a different note, an important design consideration for the parallel execution model came from the target multiprocessor class. As mentioned before, any single processor may not have enough memory to store the entire program. Parallel execution models like the *Variable Supply Model* [61] that require a complete copy at each processor are disallowed.

1.5 Target Multiprocessor Class

The target class of multiprocessors for this dissertation satisfies the following properties:

- There are an arbitrary, finite number of identical MIMD (multiple instruction stream, multiple data stream) [22] processors. No assumption is made about the speed of these processors.
- Each processor has a finite amount of local memory; there is no global (or shared) memory. No assumption is made about the memory size except that the entire database must fit in the collection of memories of the processors in the system. The database is distributed over the processors. Parts of the database may be replicated.
- Processors are connected with some interconnection topology. They can communicate only by sending messages to each other.
- Message delay is some function of the amount of data in the message and the distance between source and destination. In general, if the source and destination are not identical, there will be some non-zero delay.

- Each processor can perform backward-chaining deductions based on the subset of the database that it contains.

An architecture that satisfies the multiprocessor scenario described above is FAIM-1 [18,68].[1] Quoting from one of the papers, the FAIM-1 architecture is claimed to be "consistent with high performance VLSI implementation and packaging technology, and is easily extended to include arbitrary numbers of processors". Another architecture that would fit the requirements is the Cosmic Cube [56].

Multiprocessors that do not fall in this class are the Encore Multimax [46] and the Connection Machine [35]—the Multimax because it is a shared-memory machine and the Connection Machine because it is a SIMD (single instruction stream, multiple data stream) machine. However, it may be possible to make shared-memory multiprocessors like the Encore Multimax [46] behave like message-passing multiprocessors by making appropriate changes to the operating systems.

All the experiments described in this dissertation were done using a simulation of the FAIM-1 multiprocessor. At the level of abstraction used in the simulation, the multiprocessor is composed of a variable number of homogeneously replicated processing elements connected together with a 3-axis variant of a twisted-torus. A processing element is a processor with its own local memory. A 19 processor version would have the topology shown in figure 3. The topology is called an E-3 surface because there are 3 processing elements on each hexagonal edge. For the sake of simplicity, *wrap-around* connections for just one axis are shown. In the complete topology, two extra wires are connected to each processing element on the edge. Each processor ends up having 6 connections to its neighbors and a completely identical topological view of the rest of the processors. Quoting from the paper by Stevens [68], "this folding scheme results in ... a provably minimal diameter for hexagonal meshes." Another good feature of this topology is its scaleability. The number of processing elements on a surface is given by $3E(E-1)+1$, where E represents the E-size, or the number of processing elements on each edge. Therefore, the numbers of processors on different sizes of surfaces can be 1,7,19,37,61 and so

[1] We are assuming, of course, that each processor will have the appropriate software to do backward-chaining deductions.

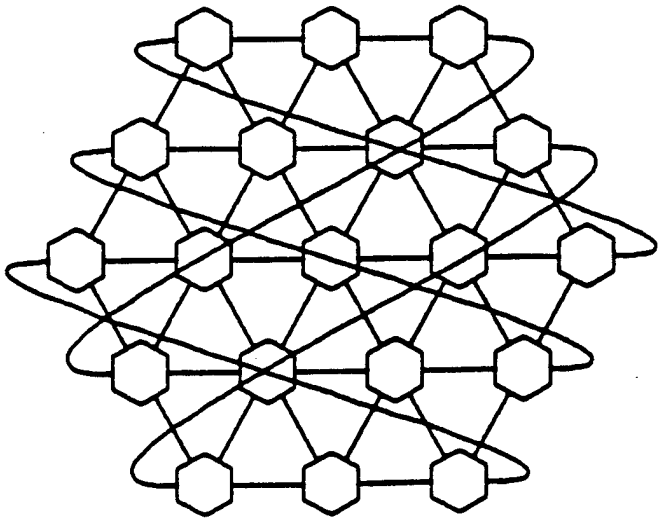


Figure 3: E-3 Processing Surface for FAIM-1

on.

The FAIM-1 multiprocessor has not been built yet but some rough estimates of its expected performance and configuration are given below. Each processor is medium-grained, larger than a Connection Machine [35] processor but smaller than a Symbolics 3600 workstation [44]. Each processor in the FAIM-1 multiprocessor is expected to perform at 20 KLIPS (1 KLIPS = 1 thousand logical inferences per second). Each processor will contain approximately 5 megabytes of memory distributed over several specialized memory types. Communication delay is expected to be $(2 + 2n + d)$ microseconds, where n is the number of packets in the message and d is the distance in hops from the source of the message to its destination. The packet size is 8 words and a word is 24 bits wide.

1.6 The Allocation Problem

We will assume for now that the computation is represented by a directed, acyclic graph (or DAG). Semantically, the graph is a dataflow graph with two exceptions. First, indeterminate merges are allowed. Second, the nodes may have associated

local state and may manipulate this local state. However, in keeping with dataflow semantics, all computation is data-driven (i.e., triggered off at nodes by messages received along the arcs). This type of graph will be called a dataflow* graph in this thesis. The name indicates the similarity to dataflow and the "*" indicates that it is slightly different from dataflow. It will be shown in chapter 2 that *PM*, the parallel execution model, is based on dataflow* graphs.

The allocation problem can be defined precisely now. It is *finding the many-to-one mapping from the set of nodes in the dataflow* graph to the set of processors that gives the minimum completion time.*

Since the precedence constraints associated with the computation DAG can be arbitrary (as can be seen later in chapter 2), this allocation problem is NP-complete because a known NP-complete problem, namely Precedence Constrained Scheduling [27], is a special case (in which communication delays are assumed to be zero). It turns out that even more structured computations are NP-complete [43]. In any case, the implication for this thesis is that finding the optimal solution is impractical. Therefore, the allocation strategy suggested by this thesis is sub-optimal. However, the allocation algorithms used are shown to be polynomial-time in their worst case complexity. Yet, the allocations generated are found to exploit much of the parallelism present in the logic programs.

In chapter 2, it will be seen that each node in the dataflow* graph is associated with a certain subset of the database, where the set of subsets is mutually exclusive and exhaustive. We will use the term *partition* for each of these subsets although this use of the term is a bit non-standard. Instead of thinking of the allocation in terms of mapping nodes of the dataflow* graph to processors, we can think of it as mapping partitions of the database to processors. Some partitions may be replicated for additional parallelism. Therefore, the mapping of database partitions to processors will be many-to-many in general.

1.7 Allocation Strategy

The allocation strategy described in this thesis is a *compile-time* (or *static*) allocation strategy. In other words, the compiler makes the decisions involved in mapping tasks to processors. This strategy is in contrast to (1) *run-time* (or *dynamic*) allocation, in which the run-time or operating system performs the allocation or reallocation, or (2) *programmed* (or *user-defined*) allocation, in which the user specifies the allocation. Compile-time allocation is not expected to be the best solution for all applications but it does compare favorably to the other two types in some ways. The disadvantage of run-time allocation is that the overhead is paid at run-time and it may be unacceptable. However, if the program behavior is highly dynamic and is hard to predict at compile-time, this may be the best approach. The disadvantage of programmed allocation is that it places a big burden on the user and the allocation probably has to be repeated for every new machine architecture. The advantage, of course, is that the user may know much more about his program and how to allocate it than an automatic allocator. Of course, features of all three types of allocation may be combined. Given that so little is known about practical allocation strategies, and almost nothing about hybrid strategies, this thesis concentrates on pure compile-time allocation. For logic programming, in particular, I do not know about any work on compile-time allocation so far.

The (possibly) limited memory size of a processor affects the resource allocation strategy also. Allocation strategies like the one described by Sarkar [55], which depend on each processor being able to execute the entire program, are unacceptable here.

The allocation strategy described in this thesis needs some restrictions that *PM* does not require. First, the type of backward-chaining deduction is restricted. In particular, no recursive clauses are allowed, unit clauses must be ground, and certain probabilistic uniformity and independence assumptions must apply. Second, a partitioning of the database is assumed to be given.

Figure 4 gives a high-level view of the allocator strategy. There are two main

modules. One module, called the allocator module, performs the search for a suitable allocation. Of course, since the search space is exponential, only a small part of it can be explored. The other module, called the cost computation module, computes the cost of a particular allocation being considered. *Cost* is a number that captures the relative poorness of an allocation.

The cost function is formally defined and domain-independent (or application-independent).[2] All the domain-dependent information required is given in the input *Goal and Domain sizes* and will be described in more detail in chapter 3. Also, the cost function does not apply just to a specific multiprocessor. The multiprocessor description is one of the inputs of the cost computation module. Again more detail is given in chapter 3. The cost function has two other important attributes. First, in an intuitive sense, the cost metric correlates well with intuitive notions of the relative poorness of allocations. This intuition is justified by experimental results obtained from an implementation of the allocator. Second, the algorithms to compute this cost function have polynomial-time worst-case complexity in the size of the computation. An exponential-time complexity would be considered unacceptable.

The allocator module consists of two phases: (1) a greedy allocation phase and (2) a local minimization phase. Let us assume for now that each partition of the database is allocated to a single processor. The greedy allocation phase allocates the partitions of the database one at a time, allocating the latest partition to the least cost processor without re-allocating previously allocated partitions. This phase has polynomial-time worst case complexity. This is followed by the local-minimization phase. In this phase, partitions of the database may be re-allocated to neighboring processors if that reduces the cost. Let a *round* consist of a (possible) single re-allocation of each part of the program. Each round has polynomial-time worst case complexity. Obtaining a local minimum of the cost-function may take an exponential number of rounds, however. Fortunately, it turns out that the greedy

[2] As used here, the term *domain-independence* means that the definition of the cost-function and the algorithms to compute it are the same regardless of the domain. However, certain inputs to the cost-function and the associated procedures may depend on the domain of interest. Smith [65] prefers to call this *semi-independence* saving the use of *independence* for cases where absolutely no domain dependent information is used.

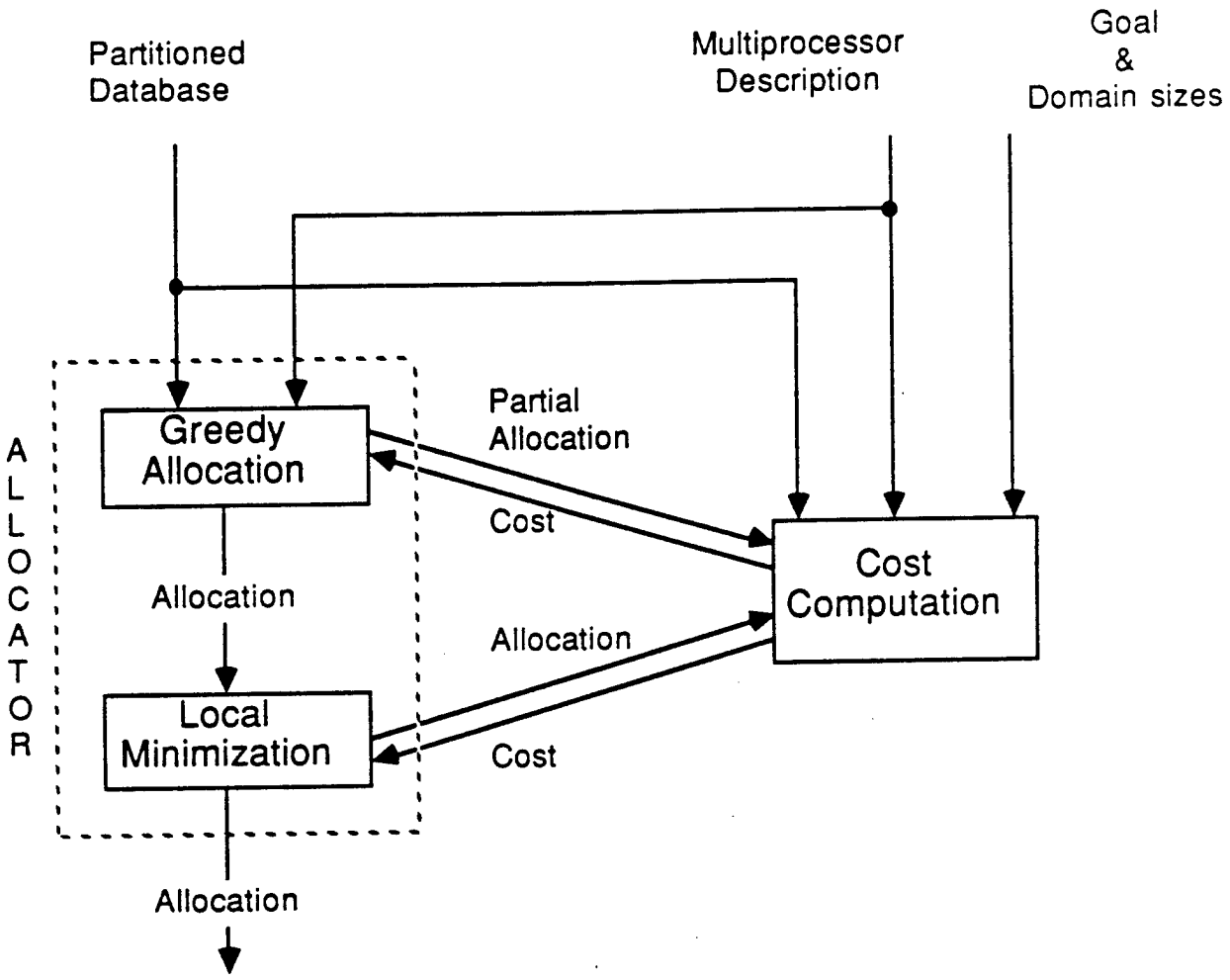


Figure 4: Allocator Strategy

allocation phase alone, or greedy allocation combined with a limited number of rounds of the local minimization phase, produces very reasonable allocations.

1.8 Organization of Document

Chapter 2 describes *PM*, the parallel execution model. Chapter 3 describes the cost-function that is the basis of the allocator. The chapter includes descriptions of algorithms to compute the cost-function and re-compute it for small changes in the

allocation. Chapter 4 describes the algorithms for allocation. The chapter includes results obtained from implementations of *PM* and the allocator. Finally, chapter 5 presents a summary of the key ideas in this thesis and directions for future research.

Chapter 2

PM: A Parallel Execution Model

2.1 Introduction

The parallel execution model described in this chapter is called *PM*. It is designed to exploit parallelism for backward-chaining deduction. In addition, *PM* is designed for a class of multiprocessors that includes non-shared memory among other features (see chapter 1 for more details). Side-effects to the database of facts and rules are not allowed during the computation in *PM*.

A key feature of *PM* is that all control of execution is based on what we call dataflow* graphs. These are dataflow graphs [70] augmented with two non-dataflow features—indeterminate merge and local state. Dataflow* carries with it the dataflow advantage of decentralized control. No synchronization is required other than the flow of data.

Several important types of parallelism have been identified for backward-chaining deductions [15,57]. The three that are exploited by *PM* are *and-parallelism*, *or-parallelism*, and *pipelining*. *Or-parallelism* is the simultaneous exploration of multiple paths to solving a single goal. *And-parallelism* is the simultaneous solution of multiple parts of a conjunctive goal. *Pipelining* also applies to the solution of constituent conjuncts in a conjunctive goal. It is the continuous streaming of solutions between a pair (or more) of conjunct solvers in sequence. Just as in pipelined

computer architectures, pipelining can improve the throughput of processing.

Unrestricted *and-parallelism* is usually not exploited because of its wasteful, combinatoric explosion. Various researchers have considered different methods of restricting *and-parallelism* [57,15,19,61,41]. The *and-parallelism* exploited by *PM* is of the type described by Conery [15], where conjunctive goals are not solved in parallel if they share variables.

Conery's execution model exploited a combination of *or-parallelism* and *and-parallelism* [15]. Lindstrom et al. [41] and I [61] used a combination of *or-parallelism* and *pipelining*. *PM* is unique in exploiting all three together for the class of architectures described above while still using data-driven control.

Resource allocation techniques are needed to determine (1) the distribution of the database over the processors and (2) the processor to use in the case of replication of certain parts of the database. Clearly, this will strongly affect the efficiency of backward-chaining deductions. Chapters 3 and 4 will describe a specific resource allocation strategy for *PM*.

This chapter is organized as follows. First, the general approach towards exploiting parallelism is described in section 2.2. Next, *PM*, the parallel execution model advocated by this chapter is described in section 2.3. This section begins with an abstract description of *PM*, along with a proof of correctness, before plunging into some algorithmic details. Section 2.4 presents some extensions to the basic execution model. Finally, section 2.5 discusses some related work done by others.

2.2 The Approach

Section 1.2 described the standard sequential approach to backward-chaining deduction. This section describes how the sequential execution model may be changed to exploit parallelism.

Many different parallel interpretations of the and-or tree are possible. One could, of course, do everything in parallel. All or-nodes that are the children of an and-node can be solved in parallel (or-parallelism) and all and-nodes that are the children of an or-node can be solved in parallel (and-parallelism). For and-parallelism, this

would mean running a process for each of the conjuncts in parallel. This would generate many solutions, most of which might fail if there were shared variables in the conjunct that must be simultaneously satisfied. Therefore, in general, it is a good idea to avoid this highly combinatoric explosion.

The solution adopted here is to exploit all the or-parallelism but to take a more conservative position with respect to and-parallelism. Only those and-nodes that do not share any common variables are solved in parallel. Assume for now (until section 2.4 on extensions to the basic execution model) that the solution of an and-node binds all the variables in the associated literal to ground terms (i.e., terms with no variables). Once and-nodes bind certain variables, then other and-nodes may stop sharing unbound variables and those nodes can then be solved in parallel. One can think of the and-nodes as being arranged in a directed, acyclic graph (DAG). Notice that each application of a rule in the database produces one such DAG. There is a one to one correspondence between the literals in the body of the rule and the nodes in the DAG. Two examples that satisfy the constraint described above are shown for the same conjunctive goal in figure 5.

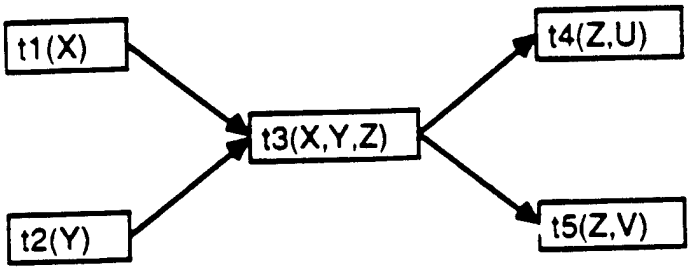
Solutions from nodes flow to their downstream neighbors which can then be solved in parallel. Solutions are sent in a continuous stream in contrast to the backtracking control of sequential and most parallel execution models. This is the essence of pipelining.

In general, some possible DAGs for a rule application will be solved more efficiently than others. In fact, this problem is analogous to ordering conjuncts for efficient sequential interpretation [64]. This problem is important but is not the subject of this thesis. In this thesis, a heuristic algorithm selects the DAG at run-time. The algorithm is described in appendix A. The input to the algorithm is a total order for a set of conjuncts—just as one would specify in Prolog, for example. The partial order generated by the algorithm is a minimal subset of this total order satisfying the constraint that conjuncts sharing unbound variables must be solved sequentially. Note that the chosen DAG is, in general, different when different sets of variables get bound at rule application time. In addition, the specific DAG representation of the partial order is minimal (in the number of edges used). The

Rule h(X,Y,Z,U,V) :- t1(X), t2(Y), t3(X,Y,Z), t4(Z,U), t5(Z,V)

Goal t1(X), t2(Y), t3(X,Y,Z), t4(Z,U), t5(Z,V)

DAG 1



DAG 2

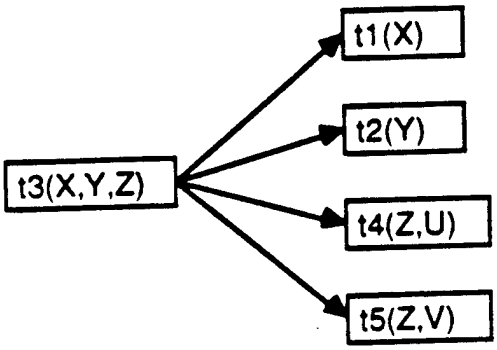


Figure 5: Example DAGs

complexity of the algorithm is $O(n^3)$, where n is the number of and-nodes.

The database is distributed to the processors in the system according to three constraints. First, the set of clauses must be partitioned into mutually exclusive and exhaustive subsets such that each literal goal generated during backward-chaining can be reduced by a single subset. A partition that satisfies this constraint is simply a partition based on predicate symbols (of facts and consequents of rules). Of course, other partitions may be possible as well. Second, each subset, in its entirety, must be separately resident in the memory of one or more processors. Third, the distribution of the database is done completely before any goal is presented to the system. (There is no reason why run-time distribution of the database cannot be done. It is just that it is not explored in this thesis.)

2.3 Basic Execution Model

The basic execution model deals with a simplified view of the multiprocessor environment as well as of backward-chaining. The additional complexities are handled in the extensions to the basic execution model.

The simplifications are as follows: (1) It is assumed that the set of clauses pertinent to reducing any particular goal are in a single processor. For example, if facts are partitioned on the basis of predicate symbols, all facts with a certain predicate symbol are in a single processor. (2) It is assumed that once the database is distributed over the multiple processors, there is no shortage of dynamic storage at individual processors during the computation.[1] (3) Finally, it is assumed that all solutions to a goal bind all the variables in the goal to ground terms (i.e., terms not containing any variables).

[1]It can be argued that this simplification violates the assumption of limited memory at each processor. In general, it is impossible to guarantee, even for sequential computations, that the amount of dynamic memory is sufficient for the given computation. In specific cases, for both sequential and parallel computations, it may be possible to guarantee that the amount of memory is sufficient.

2.3.1 Notation and Definitions

Let $< E_1, E_2, \ldots, E_n >$ denote a tuple of elements $E_1, E_2, \ldots, E_n$.

Let $\{E_1, E_2, \ldots, E_n\}$ denote a set of elements $E_1, E_2, \ldots, E_n$.

Bindings of variables are given as $Variable1 = term1$. Unification of two literals may result in a substitution given by a set of bindings. For example,

$$Substitution1 = \{V2 = term2, V3 = term3\}$$

The domain of a substitution is defined to be the set of variables whose bindings are given in the substitution. For example, the domain of the substitution $\{V2 = term2, V3 = term3\}$ is $\{V2, V3\}$. Similarly, the range of a substitution is defined to be the set of variables that appear in the bindings of the domain variables. For example, the range of the substitution $\{V2 = V3, V4 = V5\}$ is $\{V3, V5\}$.

Two substitutions may be composed to produce a single substitution. For any two substitutions, $S1$ and $S2$, $Composition(S1, S2)$ is defined only if the following two conditions hold: (1) The intersection of the domains of $S1$ and $S2$ is the null set and (2) The intersection of the range of $S2$ and the domain of $S1$ is the null set. In particular, what is allowed is for the domain of $S2$ to contain some variables belonging to the range of $S1$. For example, the two conditions are satisfied for the following case:

$$S1 = \{X = Y, U = V\}$$

$$S2 = \{Y = P, V = Q\}$$

When the two conditions are satisfied, the *Composition* function is simply the union function for sets. In the example, the composition would be $\{X = Y, U = V, Y = P, V = Q\}$. Also, two substitutions, $S1$ and $S2$ are equivalent if $S2$ can be obtained from $S1$ by replacing the binding of a variable belonging to $S1$, $Var1 = term1$, by $Var1 = term1\ |_{binding2}$, where $binding2$ belongs to $S1$ and $|_{binding2}$ indicates the application of binding $binding2$. For example, $\{X = Y, U = V, Y = P, V = Q\}$ is equivalent to $\{X = P, U = Q\}$.

2.3.2 Behavioral Description

This section contains an abstract behavioral description of the basic execution model. The next section contains a proof of correctness of this description. As will be pointed out later in detail, extra structure will be added to this description to make it more suitable for an implementation. It is in this spirit that we will treat *streams* of messages as *sets* of messages (without an ordering) in this section and in the next one.

The basic computation unit is a sequential process. Processes contain state and they are connected together by communication channels (abbreviated channels). Communication between processes takes place by sending a set of messages across each channel. Channels are directed. All messages that are sent at one end of a channel must arrive at the other end. Due to the correspondence between processes and channels with nodes and arcs respectively in a directed graph, the pairs of terms process/node and channel/arc will be used interchangeably in the rest of this chapter.

Parallelism in the basic execution model is achieved by running different processes in different physical processors. Of course, more than one process may be mapped to the same processor due to resource constraints and communication requirements. The details of setting up processes on different processors will be described in the section on algorithmic details (section 2.3.4).

We use the phrase *behavioral description* to denote a set of functions that take inputs and the current state as arguments and return outputs and a new state. A set of functions is needed because different types of incoming channels need different functions.

A very high level description is given now for the parallel computation, with more details given in later paragraphs. There are three types of processes (represented by boxes) and six types of channels (represented by directed arcs) as shown in figure 6. All messages on all channels consist of a single substitution each. Each Normal process is responsible for solving one literal for a set of substitutions S_1. S_1 is a function (to be described later) of the sets of substitutions that are received along the Input channels of the Normal process. All solutions of the literal for

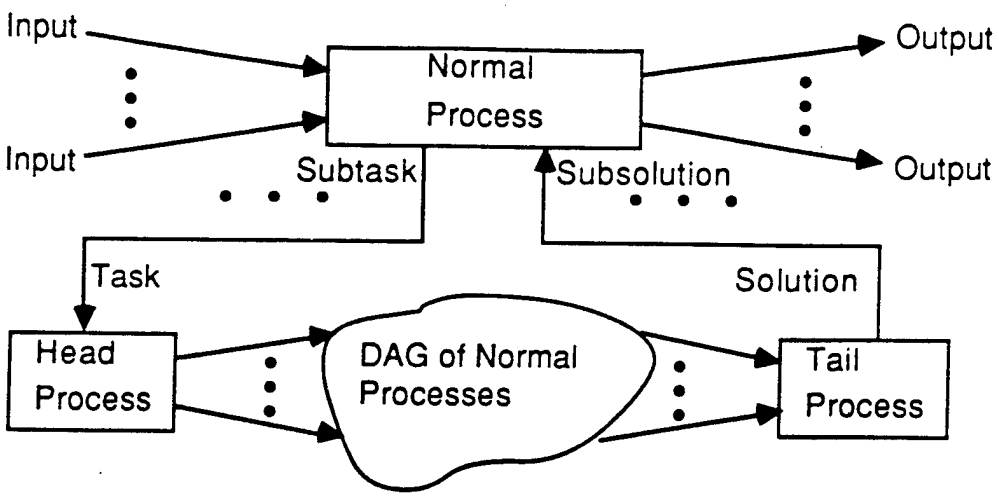


Figure 6: Types of Processes and Channels

the S_1 substitutions are sent out on each of the Output channels of the Normal process. These solutions are obtained by the reduction of goals, represented by the application of substitutions in S_1 to the literal, by rules or facts. If a rule is used, a DAG of conjunctive subgoals may be obtained of the type shown in figure 5, each conjunct being represented by its own Normal process. The Head and Tail processes shown in figure 6 are used just for the initiation of computation associated with the DAG and the collection of solutions from the DAG. If a fact is used instead of a rule, one can just think of the DAG as being empty and the Head and Tail processes as being directly connected to each other.

Other than Input and Output channels, there are Task, Subtask, Solution and Subsolution channels. A process can have at most one Task channel or Solution channel. Also, each Subtask channel has a corresponding Subsolution channel. In addition, no single process can have all types of channels. A Normal process, as shown in figure 7, can have Input, Output, Subtask, and Subsolution channels only. A Head process can have a single Task channel and some Output channels only as shown in figure 8. A Tail process can have some Input channels and a single Solution channel only as shown in figure 9. Each channel has a dual purpose when viewed from the perspective of the two processes it connects. In particular, the dual types have to be one of Input/Output, Task/Subtask, or Solution/Subsolution.

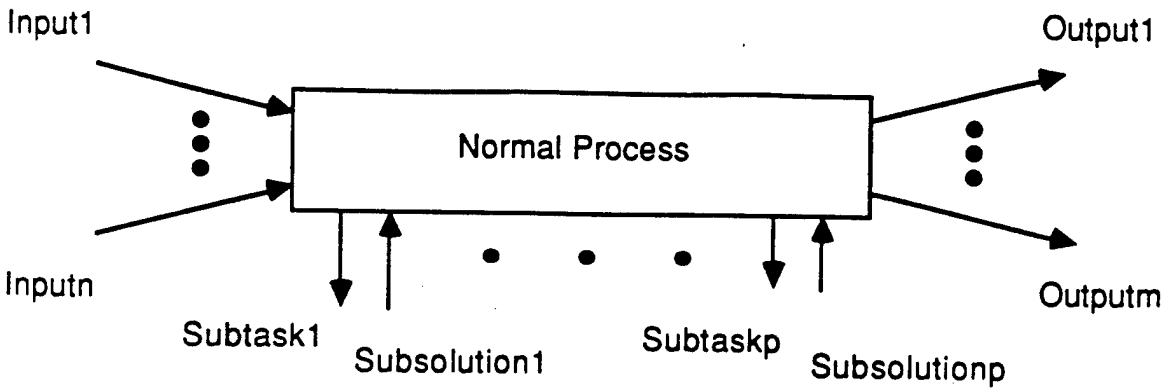


Figure 7: A Normal Process

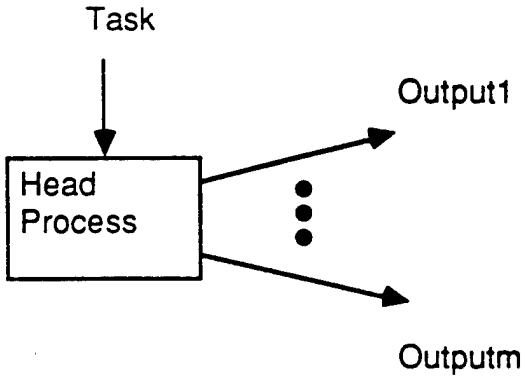


Figure 8: A Head Process

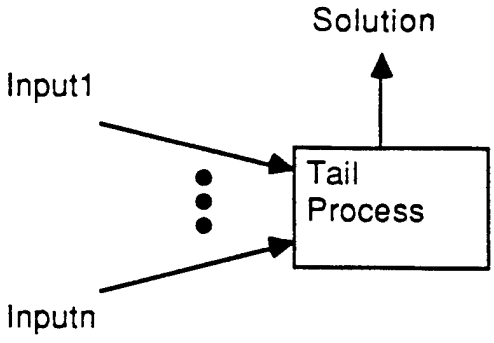


Figure 9: A Tail Process

As mentioned before, the substitutions on the input channels to a normal process represent goals that the process must solve. In particular, all input channels to a normal process are functionally equivalent to just one hypothetical channel called the *virtual input channel.* Each substitution in the virtual input channel, when applied to the literal associated with a normal process, represents a goal that the process must solve. The set of substitutions in the virtual input channel is obtained by applying a function called CP to the sets of substitutions in the multiple input channels. Informally, CP computes the cartesian product of the sets of substitutions on the input channels and filters out inconsistent combinations of substitutions. The need for this filtering can be seen in figure 10. The binding of variable "X" in a substitution along the first input channel to process "d(X,Y,Z)" may be inconsistent with the binding of "X" in a substitution along the second input channel. This combination should be filtered out.

The formal definition of CP is given now enclosed by the labels *Begin formal definition of CP* and *End formal definition of CP.* Readers satisfied with the informal definition of CP given above may skip this detail safely.

Begin formal definition of CP

The formal definition of CP uses an auxiliary function $Merge$. The input to $Merge$ is n substitutions $IS_1, IS_2, ..., IS_n$. The output is a substitution or a special element $\perp$ that is not a substitution. If there exists some variable V such that its binding in IS_i $(1 \leq i \leq n)$ is $V = b_i$ and its binding in IS_j $(1 \leq j \leq n)$ is $V = b_j$ and $b_i \neq b_j$, then $Merge(IS_1, IS_2, \ldots, IS_n) = \perp$. Otherwise, $Merge(IS_1, IS_2, \ldots, IS_n) = Union(IS_1, IS_2, \ldots, IS_n)$. $Union$ is the normal set union. The element $\perp$ is used to indicate that inconsistent bindings of some variable exist in the substitutions. This is used in the definition of CP to filter out such combinations of substitutions. A couple of examples of $Merge$ are given below.

$$Merge(\{X = x1, Y = y1\}, \{X = x1, Z = z1\}) = \{X = x1, Y = y1, Z = z1\}$$

$$Merge(\{X = x1, Y = y1\}, \{X = x2, Z = z1\}) = \perp$$

Note that all bindings are to ground terms as assumed earlier.

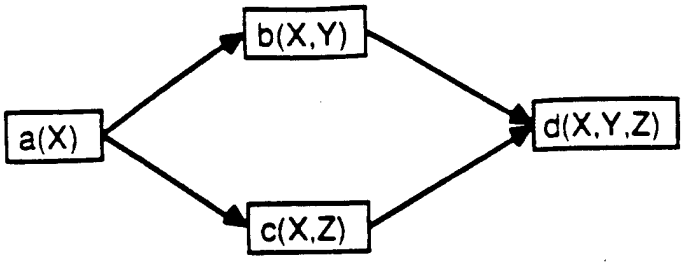


Figure 10: Filtering Substitutions

The input to the function CP is n sets of substitutions $ISS_1, ISS_2, ..., ISS_n$. The output is a set of substitutions.

$$CP(ISS_1, ISS_2, ..., ISS_n)$$

$$= \{Merge(e_1, e_2, \ldots, e_n) \mid e_1 \epsilon ISS_1, e_2 \epsilon ISS_2, \ldots, e_n \epsilon ISS_n\} - \{\perp\}$$

"$-$" is used to denote *set difference.*

As a specific example,

$$CP(\{\{X = x1, Y = y1\}, \{X = x2, Y = y2\}\}, \{\{X = x2, Z = z1\}, \{X = x2, Z = z2\}\})$$

$$= \{\{X = x2, Y = y2, Z = z1\}, \{X = x2, Y = y2, Z = z2\}\}$$

End formal definition of CP

We have seen that the set of substitutions in the virtual input channel is obtained by applying the function CP to the sets of substitutions on the input channels. It is in this sense that a single virtual input channel is equivalent to the multiple input channels to a normal process. Therefore, without loss of generality, we can complete the behavioral description of a normal process assuming just one input channel—the virtual input channel.

Just as a normal process can have more than one input channel, it can have more than one output channel. The messages on all output channels are identical. Therefore, in addition to assuming just one input channel, we can assume just one (virtual) output channel to complete the behavioral description without loss of generality.

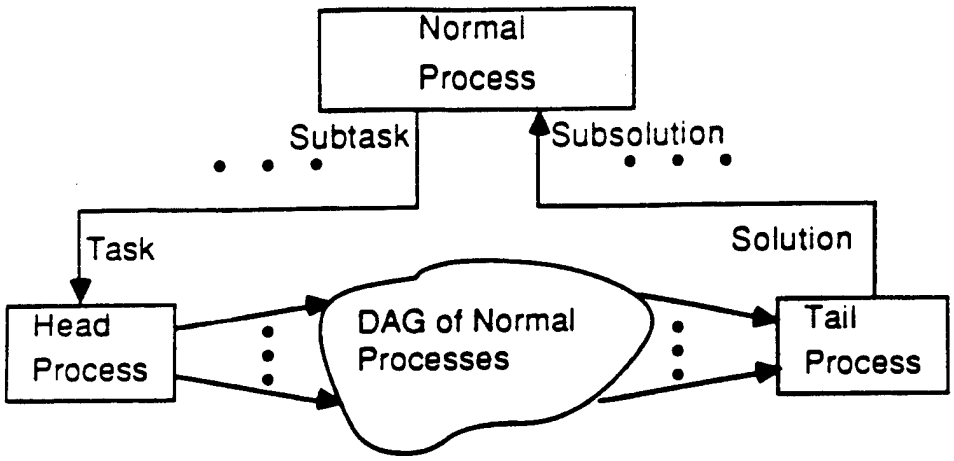


Figure 11: Reduction of Goals

As mentioned before, each substitution in the virtual input channel applied to the literal associated with a normal process is an input goal for the process to solve. The solution to each goal is also represented as a set of substitutions. The set of substitutions in the output channel is the union of the sets of solutions of the input goals.

First, consider the case when the logic program contains only assertions to solve a particular input goal for a normal process. In this case, the goal can be solved immediately and sent out on the output channel of the process.

When the logic program also contains rules, additional computation needs to be performed. All solutions found by using assertions are immediately sent on the output channel as before. For each rule that can be used to reduce the goal, unification is attempted between the goal and the head of the rule. If unification fails, nothing further needs to be done for this goal/rule pair. If unification succeeds, the substitution used for the unification is used to create a subgoal. The subgoal is simply the substitution applied to the tail of the rule. A matching pair of subtask/subsolution channels is created for the process as shown in figure 11. The input substitution that created the goal is kept in the process as state to be used later. The subtask channel carries just one element, an empty substitution, to start the solution of the subgoal. The subsolution channel brings back a set of solutions to the subgoal.

To make the solution of the subgoal possible, a two-terminal DAG of processes is set up between the subtask and subsolution channels. The graph is called two-terminal because it has two special nodes, an input node and an output node. The input node contains arcs to all nodes without any other inputs and the output node contains arcs from all nodes that do not have any other outputs. In our case, the input and output nodes are the Head and Tail nodes respectively. The DAG between the Head and Tail nodes is of the type shown in figure 5 for conjunctive goals. The DAG corresponds to the conjunctive goal that is obtained by instantiating the tail of the rule with the unification substitution. An example of such a two-terminal DAG is shown in figure 12. Notice that variables U and V have been renamed to U101 and V102. In fact, all variables in the rule must be "standardized apart" before unification [48]. When the subgoal graph is set up, a piece of state, called the *Invocation-Substitution*, needs to be stored in the Tail process. This is the subset of the substitution (resulting from unifying the goal with the rule) that contains bindings of variables in the goal (i.e., bindings of variables in the rule are ignored). Figure 12 shows the *Invocation-Substitution*, shown as IS in the figure, that needs to be stored for the example. Notice that this design decision leads to what might be called *distributed binding environments*. An alternative might have been to copy the complete environment and send it to the processes associated with the subgoal graph. However, the problem with copying is that the environments might get very large and the messages containing them may have excessive communication delays.

The top level goal to the system is also represented like any other subgoal in the system (i.e., it is a two-terminal DAG of processes). For the top level goal, the *Invocation-Substitution* is empty. A top level goal is shown in figure 13. "{{}}" next to the task channel of the Head process indicates that the set contains just one element, an empty substitution.

The purpose of the Head and Tail processes needs to be explained now. Both are not associated with any literal.

The Head process merely serves as a router of data. When it receives an empty substitution along its subtask channel, it sends copies of the same on all its output channels.

Rule a(W,x1,Y,Z) :- b(W,Y),c(Y,U),d(Y,V),e(U,V,Z)

Goal a(w1,X,Y,Z)

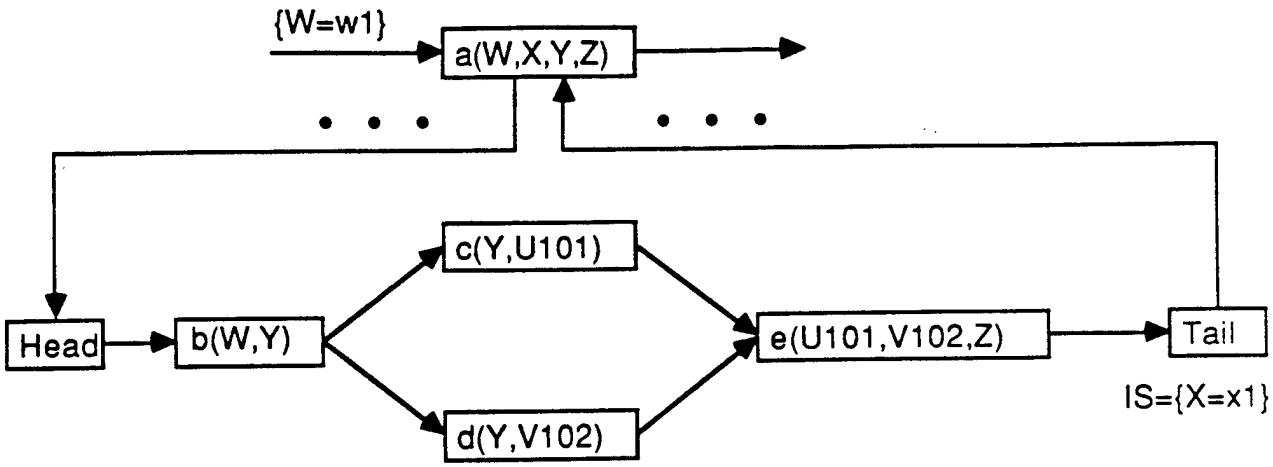


Figure 12: Example of Goal Reduction

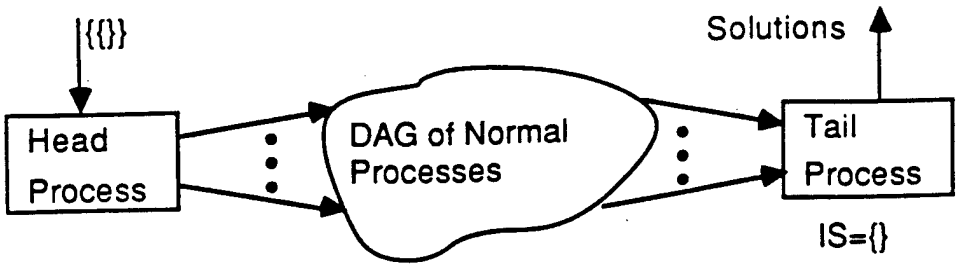


Figure 13: Top Level Goal

As mentioned above, the Tail process stores an *Invocation-Substitution* in its state. The Tail process receives substitutions along its input channels and it computes the cartesian product of the associated sets of substitutions like any other normal process. The rest of its behavior is different from a normal process. For each substitution on its virtual input channel, it sends a substitution on its solution channel. The solution substitution is created by applying the *Composition* function to the *Invocation-Substitution* and the input substitution. As an example, consider figure 12 again. If the Tail process receives the input substitution {Y=y1,U101=u1,V102=v1,Z=z1}, then the corresponding solution substitution is {X=x1,Y=y1,U101=u1,V102=v1,Z=z1}.

A normal process may have several subsolution channels, one for each of the subgoals created. The input substitution used to create the goal is kept as state in the process. When the process starts receiving substitutions along its subsolution arcs, the following is done for each substitution: The *Composition* function is applied to the associated input substitution and the subsolution substitution. The resulting substitution is sent out on the virtual output channel. Subsolution substitutions are processed in this manner as they arrive. If the order of arrival cannot be determined (when they arrive too close to resolve the difference in times), then they are processed in an indeterminate order. It is in this sense that we can say that the output channel of the process is created from the indeterminate merge of the solutions of its subgoals. As an example, consider figure 12 again. If a subsolution substitution for the given goal is {X=x1,Y=y1,U101=u1,V102=v1,Z=z1}, then the corresponding output substitution is {W=w1, X=x1, Y=y1, U101=u1, V102=v1, Z=z1}.[2]

The graph that is generated in the process of goal reductions starting from the top-level goal is the dataflow* graph for the computation. Note that this graph is not present before run-time. Also, there is no need to have an explicit representation of this graph at run-time. However, algorithms, presented later in chapters 3 and 4,

[2]Clearly, the bindings for variables U101 and V102 are not necessary. If required, these could have been pruned either by the Tail process or the Normal process. The current implementation leaves these bindings in because they provide useful information during program development. A production system should prune these bindings if its only goal is efficiency.

will be used to predict certain properties of these graphs for the purpose of resource allocation.

2.3.3 Proof of Correctness

Theorem 1 *For deductions with a finite and-or tree, the set of solutions produced by PM is equal to the set of solutions produced by a Prolog interpreter.*

Notice that the Prolog interpreter was defined in section 1.2. To prove the theorem, we will prove two lemmas first. Before we get to the lemmas, a few definitions need to be stated.

For a directed graph, a node N1 is defined to be a direct predecessor of node N2 if and only if there is an edge from N1 to N2. Similarly, a node N1 is defined to be an ancestor of N2 if and only if N1 is in the transitive closure of the direct predecessor relation of N2. If a directed arc goes from node A to node B, A is called the source node and B is called the destination node of the arc. Note that "node" and "process" are used interchangeably.

Lemma 1 *For each input channel to a process P, if the set of substitutions contained in the channel is equal to the set of solutions to the conjunctive goal CG1, where CG1 is the set of the literal associated with the source process of the channel and all literals associated with the ancestors of the source process, then the set of substitutions in the virtual input channel of the process P is equal to the set of solutions to the conjunctive goal CG2, where CG2 is the set of literals associated with all the ancestors of the process P.*

Proof: The statement "For each input channel to a process P, the set of substitutions contained in the channel is equal to the set of solutions to the conjunctive goal CG1, where CG1 is the set of the literal associated with the source process of the channel and all literals associated with the ancestors of the source process" in the first part of the lemma will be referred to as the correctness condition of the lemma. Assume for now that the process in question has two input channels. The proof can be easily extended to an arbitrary number of channels by induction

on the number of channels. Let the set of literals associated with the source process of the first channel and all its ancestors be {C1,C2,...,Ci,Ci+1,...,Cm} and the corresponding set for the second channel be {Ci,Ci+1,...,Cm,Cm+1,...,Cn}. Call these two sets A and B respectively. Notice that the two sets have an arbitrary set of literals, {Ci,Ci+1,...,Cm}, in common. The set of ancestors of the process is given by the union of A and B, {C1,C2,...,Cn}. Call this set C. We know that the solutions to C are exactly the same as the solutions of the bag, D, containing the sum[3] of A and B considered as bags. This is true because a conjunctive goal with duplicate conjuncts is equivalent to a conjunctive goal with the duplicates removed. Therefore, the lemma is reduced to the statement that applying CP to the sets of solutions of A and B gives exactly the set of solutions to the conjunctive goal composed of A and B. This simplified statement will be proved by showing a subset relationship both ways.

First, let us prove that every solution of the conjunction of A and B is a member of the result of CP. Let us pick an arbitrary solution S1. We know that any solution of a set of conjuncts must be a solution of a subset also. (This follows easily from the definition of the Prolog interpreter in section 1.2.) Therefore, S1 must be a solution of A and it must be a solution of B. Of course, S1 may contain a superset of the bindings required for A and B separately. In addition, the correctness condition of the lemma tells us that this solution must be a member of both the input sets of substitutions to the node. Actually, only the subset of S1 relevant to A will be in the first channel. The same applies for B. If this is the case, then the definition of CP requires that the union of the two substitutions along the two channels (i.e., S1) be a member of the result of CP.

Now, let us show the reverse subset relationship to prove equality of the two sets. We need to show that every member of the result of CP is a member of the solution set of the conjunction of A and B. Recall from the definition of CP that each member of the result of CP above will be the union of a substitution from the first channel and a substitution from the second channel. In other words, each member

[3]Sum of bags is different from union of sets in the following way. The number of instances of a member of the sum is equal to the sum of the number of instances of the member in the bags whose sum is taken.

of the result of CP is a superset of a substitution on the first channel and also a superset of a substitution on the second channel. Since the correctness condition of the lemma states that each member of the first channel is a solution of A and each member of the second channel is a solution of B, each member of the result of CP is a solution to A as well as B. Therefore, it is a solution of the conjunction of A and B. (This follows easily from the definition of the Prolog interpreter in section 1.2.)□

Lemma 2 *Consider a two-terminal DAG of processes in which the input node is a Head process, the output node is a Tail process, and the DAG in between is composed of normal processes. For this graph, sending the Head process an empty substitution will produce, at the virtual input of the Tail process, the set of solutions to the conjunctive goal composed of the literals associated with the normal processes* provided *that each process individually solves the goals input to it correctly.*

Proof: The statement "each process individually solves the goals input to it correctly" in the last part of the lemma will be called the correctness condition of this lemma.

We need to define the concept of distance of a process from the Head process. Let distance of 1 denote that there is a direct edge from the Head process to the process. A distance of n indicates that the maximum distance of a direct predecessor of the process is $n-1$. Let the distance of the Tail node also represent the length of the graph. Notice that all such graphs have a finite length because they are DAGs.

Now, the lemma is trivially true for all such graphs in which the graph length is 2. In this case, there are a set of normal nodes in parallel after the Head node and there are edges from all these nodes to the Tail node. In this case, lemma 1 applies directly.

Now, the induction hypothesis is that the lemma is true for graphs of lengths up to n. The induction step requires that we prove that the lemma is also true for graphs of length $n+1$. For a graph of length $n+1$, consider all nodes that are direct predecessors of the Tail node. Replace one such node P by a new Tail node and consider the graph between this new Tail node and the original Head node.

The induction hypothesis can be applied to this graph because it has a length of n or less only. Therefore, the set of substitutions in the virtual input channel of the new Tail process is equal to the set of solutions of the associated conjunctive goal. This has an implication for the original graph. The set of substitutions in the virtual input channel of the node P (that is transformed to a Tail node) is equal to the set of solutions to the conjunctive goal represented by the set of nodes that are ancestors of the node P. Now, the virtual input channel of the node represents goals for the node. The correctness condition of the lemma states that all such goals are correctly solved. Therefore, the output channel of the node P (which is also an input channel of the original Tail process) will contain the set of solutions to the conjunctive goal of the literals represented by the node P and all its ancestors. The same can be claimed for all input channels of the Tail node. Now, we can apply lemma 1 to prove that the set of substitutions in the virtual input channel of the Tail node is equal to the set of solutions of the conjunction of all ancestors of the Tail node (i.e., all literals in the original graph).□

Proof of Theorem 1: The dataflow* graph contains some normal processes whose solutions are produced by unification with facts directly and not by reduction to a DAG of processes obtained by applying a rule. If there were no such normal processes, the associated and-or tree would be infinite and the computation would never end. Let us refer to these nodes as nodes of level 1. In general, a node is defined to be of level $n+1$ if and only if the maximum level of any node in any of its subgoal graphs is n. The maximum node level in the dataflow* graph is called the level of the graph.

The theorem will be proved by induction on the level of dataflow* graphs. The theorem is trivially true for graphs of level 1 because lemma 2 applies directly. The induction hypothesis is that it is true for graphs of level up to n. We need to show that it is true for graphs of level $n+1$. At the top level in this dataflow* graph is a two-terminal graph with some nodes of level $n+1$. For each such node, its subgoals contain nodes of level n or less. Therefore, each of its subgoals is correctly solved according to the induction hypothesis. Since the solution of the node is obtained simply by taking the indeterminate merge of the solutions of the subgoals, the node

itself is correctly solved. (Indeterminate merge produces the same solutions as the ones produced by backtracking using the Prolog interpreter defined in section 1.2.) Now, the application of lemma 2 proves that the top level DAG is also correctly solved.□

2.3.4 Algorithmic Details

The description of the basic execution model that was presented in section 2.3.2 was complete in its own right. However, modification of certain peripheral details makes the implementation easier or more efficient. In addition, it is much too abstract for a direct implementation. In this section, we describe the additional features that are added to the abstract description and then describe specific choices made in terms of state, messages, and procedures.

2.3.4.1 Additional features

There are three additional features. First, messages along channels are treated as *streams* as opposed to *sets*. Second, messages contain more than just substitutions. Third, each stream of messages is terminated by a special *end-of-stream* message.

Sets to Streams This is the most important additional feature. It is due to this feature that *PM* gets its dataflow flavor. Every channel contains a stream of messages. A stream is equivalent to an ordered set. In general, channels are not required to preserve the ordering of messages from their inputs to their outputs. Therefore, two messages that are sent in one order from the source process of a channel may arrive in another order at the destination process of the channel. Typically, messages do arrive in order. The advantage of not requiring *in-order* delivery is that message protocols can generally be simpler and faster.

Computation at processes is triggered only by the arrival of messages and by no other mechanism. In particular, complete streams need not arrive for processing to begin. In this sense, a process behaves exactly like a node in a dataflow graph. However, as noted before, processes contain state whereas dataflow nodes do not.

In general, when an input message is processed, several output messages may be generated as described in the abstract behavior. These output messages are sent out on the appropriate output channels before the next input message is processed.

The only place in the description where the order of input messages needs to be clarified is where the function CP is applied to the sets of substitutions on the input channels to produce a set of substitutions on the virtual input set. In particular, a new function $CPnew$ needs to be defined that takes n input streams of substitutions and returns one stream of substitutions to be considered the *virtual input stream.* Streams are represented mathematically as tuples. $CPnew$ is defined to be the composition of three other functions.

$$CPnew = CPnew3 \circ CPnew2 \circ CPnew1$$

$CPnew1$ takes as input n streams of substitutions and returns one stream of n-tuples (of substitutions). Let the input streams be:

$$< S_{1,1}, S_{1,2}, \ldots, S_{1,l_1} >$$

$$< S_{2,1}, S_{2,2}, \ldots, S_{2,l_2} >$$

$$\vdots$$

$$< S_{n,1}, S_{n,2}, \ldots, S_{n,l_n} >$$

The lengths of the streams are $l_1, l_2, \ldots, l_n$ as shown. The virtual input stream is specified by the elements that it contains and a total order. The elements that it contains are all $l_1 \times l_2 \times \ldots \times l_n$ n-tuples of the form:

$$< S_{1,i_1}, S_{2,i_2}, \ldots, S_{n,i_n} >$$

where $1 \leq i_j \leq l_j$. As can be seen from the prototypical tuple, its kth element comes from the kth stream for all k such that $1 \leq k \leq n$.

The order of the elements in the output stream is constrained only by a partial order to be described shortly. Since a total order is required for a stream, any particular total order that satisfies the partial order is acceptable. Two prototypical

elements $PE1$ and $PE2$ are ordered if $n-1$ of their constituent elements are the same and the nth is different. For example,

$$PE1 =< S_{1,i_1}, \ldots, S_{j,i_{j1}}, \ldots, S_{n,i_n} >$$

$$PE2 =< S_{1,i_1}, \ldots, S_{j,i_{j2}}, \ldots, S_{n,i_n} >$$

where the $S_{i,j}$ substitutions are as given above. In this case, $PE1$ will precede $PE2$ in the output stream if and only if $i_{j1} < i_{j2}$. Similarly, $PE2$ will precede $PE1$ in the output stream if and only if $i_{j2} < i_{j1}$.

As a specific example, consider the case when there are two input streams. Let one input stream be represented by the tuple $< S1, S2 >$ and the other stream by $< S3, S4 >$. There would be four elements in the output stream: $< S1, S3 >$, $< S2, S3 >$, $< S1, S4 >$, and $< S2, S4 >$. Let "$\prec$" indicate the ordering predicate. The ordering constraint described above would force the following partial order:

$$< S1, S3 > \prec < S2, S3 >$$

$$< S1, S3 > \prec < S1, S4 >$$

$$< S2, S3 > \prec < S2, S4 >$$

$$< S1, S4 > \prec < S2, S4 >$$

Therefore, the output stream is one of either

$$<< S1, S3 >, < S2, S3 >, < S1, S4 >, < S2, S4 >>$$

or

$$<< S1, S3 >, < S1, S4 >, < S2, S3 >, < S2, S4 >>$$

The motivation for this ordering constraint is that it is similar in spirit to the first-in-first-out and incremental processing that is used for a straight dataflow solution. This completes the definition of $CPnew1$.[4]

[4]Notice that, strictly speaking, one would have to remove any duplicates in the output of $CPnew1$ if one is to think of it as a set (or an ordered set). Typically, implementations of logic programming languages do not prune out duplicates in the interest of efficiency. In the same spirit, the implementation of *PM* does not remove duplicates either. Therefore, if one is to be mathematically correct, the collections of substitutions along streams should be called *bags* (or ordered *bags*).

$CPnew2$ is applied to the output of $CPnew1$. $CPnew2$ takes one stream as its input and returns one stream as its output. Each element in the input is an n-tuple of substitutions. The output of $CPnew2$ is a stream with exactly the same number of elements as the input stream. The elements of the output stream are obtained by applying the $Merge$ function (used in the description of CP) to the corresponding elements of the input stream (i.e., elements in the same positions). Note that $Merge$ takes n input substitutions and returns a substitution or a special element $\perp$. The n input substitutions in this case are the n constituent elements of each element of the input stream to $CPnew2$. The $Merge$ function is used, as before, for the purpose of filtering out bad combinations of substitutions. As an example, if the input to $CPnew2$ were

$$<< \{X = x1, Y = y1\}, \{X = x2, Z = z1\} >, < \{X = x2, Y = y2\}, \{X = x2, Z = z1\} >,$$

$$< \{X = x1, Y = y1\}, \{X = x3, Z = z2\} >, < \{X = x3, Y = y1\}, \{X = x3, Z = z2\} >>$$

then the output would be

$$<\perp, \{X = x2, Y = y2, Z = z1\}, \perp, \{X = x3, Y = y1, Z = z2\} >$$

The output of $CPnew2$ is the input to $CPnew3$. $CPnew3$ takes one stream as its input and returns one stream of substitutions as its output. The output of $CPnew3$ is exactly the same as its input except that all the $\perp$ elements are filtered out. All the non-$\perp$ elements in the input stream are retained in the output stream with the same order. As an example, if the input to $CPnew3$ were

$$<\perp, \{X = x2, Y = y2, Z = z1\}, \perp, \{X = x3, Y = y1, Z = z2\} >$$

then the output would be

$$< \{X = x2, Y = y2, Z = z1\}, \{X = x3, Y = y1, Z = z2\} >$$

The output of $CPnew3$ is the output of the top-level function $CPnew$. This completes the definition of $CPnew$. The resulting stream obtained by an application of $CPnew$ to some stream arguments will be called the *cartesian product* of the streams.

Message Content The message content is designed to not require any special messages to create processes initially. There is enough information in the messages that a process can be created when the first message for the process arrives.[5] To make this possible, each message contains more than just a substitution. In fact, each message contains a *task*. A task includes a substitution as well as a two-terminal DAG of literals representing a conjunctive goal. The two-terminal DAG for a task along an input or output channel is the subgraph that can be reached from the channel up to and including the first tail node. For example, consider figure 12 again. Tasks along the channel from the Head node to the "b(W,Y)" node would include the nodes labeled "b(W,Y)", "c(Y,U101)", "d(Y,V102)", "e(U101,V102,Z)" and the Tail node. Similarly, tasks along the channel from the node labeled "b(W,Y)" to the node labeled "c(Y,U101)" would include the nodes labeled "c(Y,U101)", "e(U101,V102,Z)" and the Tail node. The two-terminal DAG for tasks on task/subtask channels is the graph for the entire conjunctive subgoal (including the Head and Tail nodes). The two-terminal DAG for tasks on the solution/subsolution channels is empty.

End-of-Stream Message Another feature that is added in the detailed description is *end-of-stream* messages. These are special messages that are sent on streams after the last regular message has been sent. The advantage of this feature is that the top level process can tell when it has produced the last answer. This is the only place in the description of *PM* that temporal ordering of messages on streams is necessary. There are many ways of doing this with much less overhead than the case in which all messages on a stream are required to be temporally ordered.[6]

The rest of this section contains detailed descriptions of all the state, messages, and procedures required for the basic execution model.

[5] It may still be the case that additional messages to set up later processes concurrently with processing of the earlier processes in the DAG may be more efficient.

[6] For example, the end-of-stream message may include the number of messages that have been sent on the stream so far. The destination node must also keep a counter of messages received. When an end-of-stream message is received, its processing is postponed till the right number of regular messages is received first.

2.3.4.2 State

Each processor, process and task has a system-wide unique name.[7] In the rest of this thesis, typical names for processors, processes, and tasks will be of the form P_i, PS_i, and T_i respectively.

Each processor maintains the following state information on the tasks and processes for which it is responsible:

Work-Set: This is a set of tasks that the processor may work on.

Task: Each task is a 5-tuple of the form:
<Task-Name, Task-Description, Subtasks, Spawning-Process-Name, Parent-Task-Name>

Task-Name is the system-wide unique name of the task. *Task-Description* is the description of the task. This field contains the substitution that was described earlier as the sole content of a message. This field will be described in more detail below. The cartesian product of input streams of tasks produces a single virtual input stream of tasks. The *task description* field of each task in the virtual input stream gets its substitution exactly as described in the behavioral description. For each task that is generated by the cartesian product function, the *Spawning-Process-Name* field is set to the name of the process that applied the cartesian product. This field is empty for any other tasks. Again, for every task in the virtual input stream, multiple subgoals may be created by the application of rules that can reduce the goal represented by the task. The reduced goals are represented as tasks and the name of each such reduced task is a member of the *Subtasks* field of the parent task. Similarly, child tasks (i.e., tasks that are produced by the goal reduction) have their *Parent-Task-Name* field set to the name of the parent task. Variables in the rule must be "standardized apart" before unification with the goal literal.

Task-Description: Each is a tuple of the form: *<CG, BL>*

CG (or Conjunct Graph) is a two-terminal DAG. *BL* is a substitution. The nodes in the graph are processes (as specified below).

[7] All that is needed for this to work is that each processor have a unique name and each processor have a processor-wide unique name generator. Unique system-wide names for processes and tasks can now be generated by combining the system-wide unique processor name, where the process or task is to be generated, with a processor-wide unique name.

Process: Each is a 10-tuple of the form:

<Process-Name, Literal, Processor-Name, Number-Inputs, Input-Queues, Outputs, Spawned-Task-Names, Type, Child-Task-Name, Invocation-Substitution>

Process-Name is the system-wide unique name of the process. *Literal* is the literal that the process is responsible for solving. *Processor-Name* is the name of the processor where the process resides. *Number-Inputs* is the number of inputs to the process. *Input-Queues* are the queues of messages waiting to be processed at the inputs to the process. These queues contain additional state to (1) indicate whether the end-of-stream message has been received and (2) give the status of the cartesian product formation from the inputs (more later on this). *Outputs* are a set of tuples specifying the inputs of other processes. Each tuple is of the form *<process-name, processor-name, input-number>*. *Spawned-Task-Names* is a set of task names. The names correspond to tasks that are created by cartesian product. In case no cartesian product is necessary (when there is only one input), the unmodified task names from the inputs are directly included in *Spawned-Task-Names.* The *Type* of the process can be one of {Normal, Head, Tail}. The *Invocation-Substitution* has been described before.

A complete process specification as given above is not necessary for each node in the conjunct graph of a task specification. A partial specification as given below is sufficient. "*xxx*" indicates an unspecified field.

<Process-Name, Literal, Processor-Name, Number-Inputs, xxx, Outputs, xxx, Type, xxx, xxx>.

The unspecified fields are *Input-Queues*, *Spawned-Task-Names*, *Child-Task-Name* and *Invocation-Substitution* from left to right.

Notice that the *Processor-Name* field is included. In particular, this means that a process that creates a subgoal/subtask must bind all processes in the conjunct graph of the subtask to specific processors. In case multiple choices exist (when certain subsets of the database are replicated), resource allocation procedures must be invoked to make the choice.

2.3.4.3 Messages

Messages are 4-tuples of the form:
<Message-Type, Source-Processor-Name, Destination-Processor-Name, Arguments>

For now, only one message type is required. More types are required for the extensions to the basic execution model. The type needed now is *Input-Task*. For this, the *Arguments* field is a tuple of the form:
<Destination-Process-Name, Destination-Input-Number, Task-Name, Task-Description>

The fields have self-explanatory names. End-of-stream is indicated with "EOS" as the substitution in the *Task-Description*.

2.3.4.4 Procedures

As mentioned before, the database of rules/assertions is distributed before any goal is ever presented to the system. Also, all rules/assertions that can be used to reduce any particular task are in a single processor. It turns out each processor in the system need not have the complete partitioning information at run-time. Even the partitioning information may be distributed. A processor needs to know only the identity of processors that can be used to solve each literal in the tails of the rules that it contains (i.e., each literal in the conjunctive subgoals that it generates itself). The processor that is given the top level goal must know the identity of all processors relevant to solving each literal in every goal that may be presented to the system.

When a process on a processor creates a subtask, the Head and Tail processes associated with the subtask are created at the same processor. The message containing an empty substitution to the Head process can be replaced by a function call. Similarly, the output messages from a Tail process on its solution stream may be replaced by function calls since the destination of the messages is the same processor. Therefore, messages are needed along input and output channels only. Messages along other types of channels can be replaced by function calls.

As mentioned before, every task on an input/output channel contains the two-terminal DAG that can be reached from the channel. Therefore, DAGs in tasks input to a process must have their input node "stripped off" to obtain the DAGs that must be output from the process. The cost of this procedure is simply the cost of traversing the two-terminal subgraphs that can be reached from the outputs of the process.

The cartesian product function was described earlier. One interesting feature of this function is that it can be computed "incrementally". As messages arrive on the input channels of a process, they are kept in a FIFO queue. Consider the situation when there are some messages in the queues and a message arrives on one of the channels. The virtual tasks that can be created out of the combination of this task with the tasks waiting in other queues may be immediately computed. Of course, the order of these newly generated virtual input tasks on the virtual input stream must satisfy the order prescribed by the cartesian product function. Clearly, if the cartesian product function is going to be computed incrementally, then some state needs to be kept to indicate the extent to which the cartesian product has been computed at any given time.

As mentioned before, the last message on each stream is a special *end-of-stream* message. Special care must be taken to send these messages only when all other messages have been sent on a stream. In particular, a normal process will send end-of-stream messages on its output channels (one on each) when the conjunction of the following three conditions is satisfied: (1) All input channels have received an end-of-stream message. (2) All tasks on the virtual input stream have had their subtasks created. (3) All subsolution channels have received an end-of-stream message. For a Tail process, since there are no subsolution channels, condition (2) may be left out. In addition, the end-of-stream message is not sent on any output channel (since the Tail process does not have one) but it is sent on the solution channel. In the case of Head processes, only one input message is received on the task channel. Therefore, the end-of-stream need not be sent explicitly. For messages output from a Head process, each channel carries two messages exactly—an input-task message and an end-of-stream message.

C1	r(X,Y):-p(X),q(Y),s(X,Y)	P1
C2	p(a)	P2
C3	p(b)	
C4	q(Y):-m(X),n(X,Y)	P3
C5	m(a)	P4
C6	m(b)	
C7	n(b,a)	P5
C8	n(b,b)	
C9	s(a,b)	P6
C10	s(b,a)	

Figure 14: An Example Database

2.3.5 A Complete Example

As mentioned before, a dataflow* graph is the graph of process nodes that is generated during the execution of *PM*. However, just as a syntactic and-or tree is easier to view than a complete and-or tree, a syntactic version of the dataflow* graph is easier to comprehend. In the syntactic version, a process is connected (by subtask and subsolution channels) to a single copy of the subgoal graph for each rule/assertion that applies to the literal associated with the process.

Consider the example database shown in figure 14. The distribution of the database is also indicated in the figure. In the example, the database is partitioned on the basis of predicate symbols and each subset is resident on a single processor.

A graphical abbreviation is used to reduce the complexity of the dataflow* graph of the example. This abbreviation is shown in figure 15.

The syntactic dataflow* graph associated with the database for the query r(X,Y) is shown in figure 16. Solid boxes indicate processes. The literals inside the boxes are the literals to be solved by the processes. The exceptions are the Head and Tail process pairs which are shown as boxes with "H/T" inside. Dashed lines around sets of boxes indicate that those processes reside in the same processor. The name of the

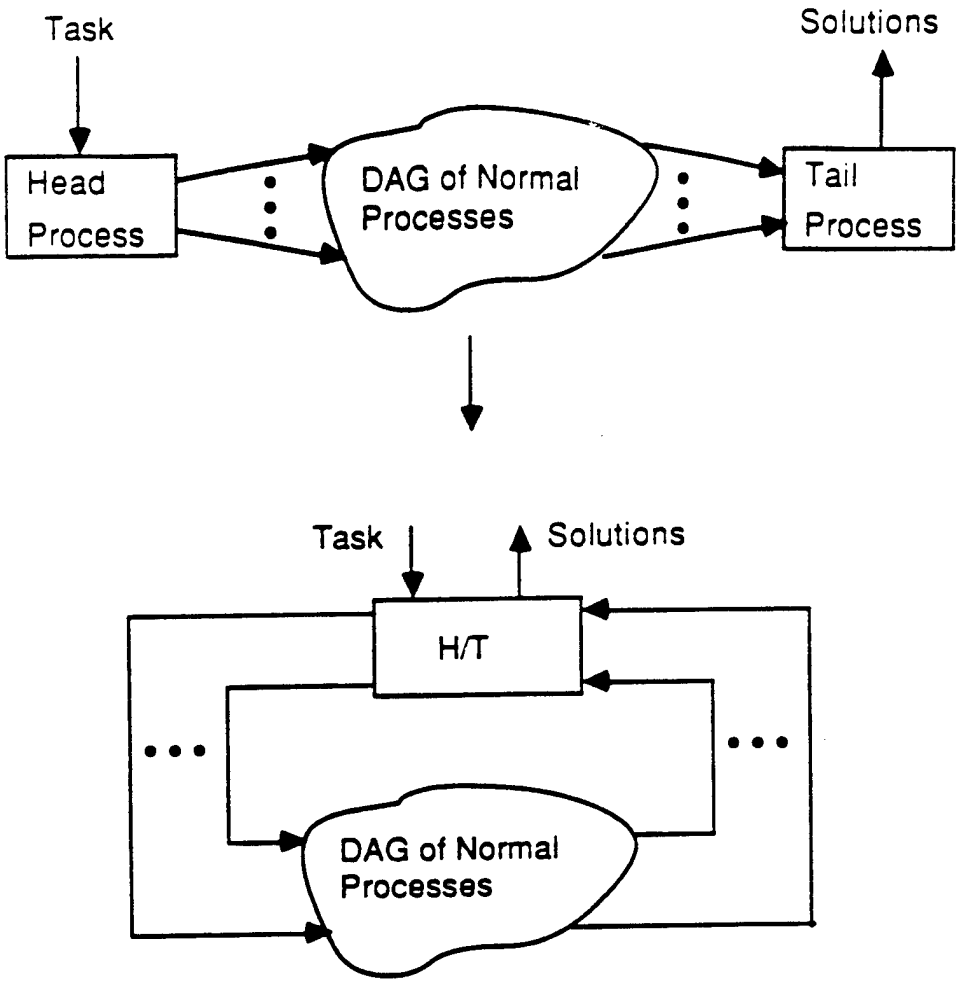


Figure 15: Graphical Abbreviation for Dataflow* Graphs

processor is indicated as a name of the form Pi. Arcs that cross dashed lines indicate streams of *input-task* messages. Task names (of the form Ti) are written next to the arcs. "," indicates temporal sequencing. Arcs inside dashed lines indicate function calls within the same processor to set up child tasks (downward arcs) and to send solutions to parent tasks (upward arcs).

The top level task is T1 at processor Pi. It turns out that it has only one literal "r(X,Y)" to solve. In general, there could be an arbitrary number.

Notice a couple of different dataflow* subgraphs for child tasks. The conjunctive goal "p(X),q(Y),s(X,Y)" leads to the conjunct graph with "p(X)" and "q(Y)" solved in parallel followed by "s(X,Y)". In the case of the conjunctive goal "m(X),n(X,Y)", the two literals must be solved sequentially because they share the variable "X".

Finally, figure 17 shows some abbreviated task descriptions. To avoid cluttering up the figure, task tuples have been abbreviated to the shortened tuples

$$<\textit{Task-Name},BL,\textit{Parent-Task-Name},CG>$$

where BL is the associated substitution and CG is the conjunct graph.

The sample task shown on top contains mnemonic field names to make it easier to decode the fields in the examples. In addition, the process nodes in the CGs (or Conjunct Graphs) are abbreviated to just the associated literals. The *Invocation-Substitution* for Tail nodes is shown directly below the boxes representing them.

T1 is the task representing the top-level goal. T2 and T3 are two solutions for the top-level task. T4 is the end-of-stream message for the solution stream associated with T1. In fact, the last task in each stream (except streams going to Head processes) is a similar end-of-stream message.

A child task such as T33 gets a variable renamed X101 uniquely because variables in rules are "standardized apart" before unification with goals.

Each process is responsible for solving the input node in the conjunct graph of each task on its input streams. The outgoing tasks are, therefore, the incoming tasks with the input node "peeled off". For example, look at task T47 and task T49. In general, "peeling off" the input node of a task can create multiple tasks.

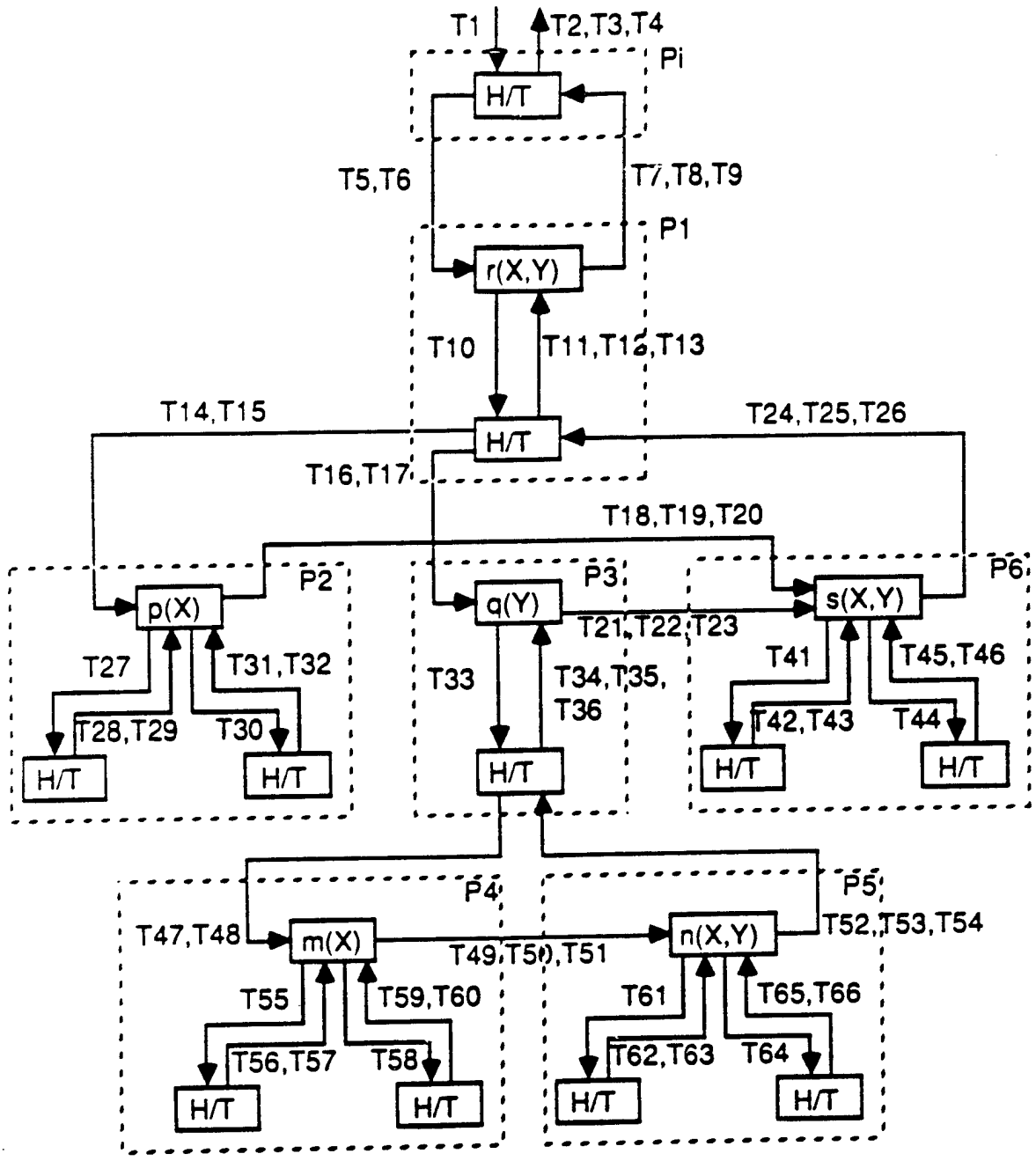


Figure 16: Dataflow* Graph for Example

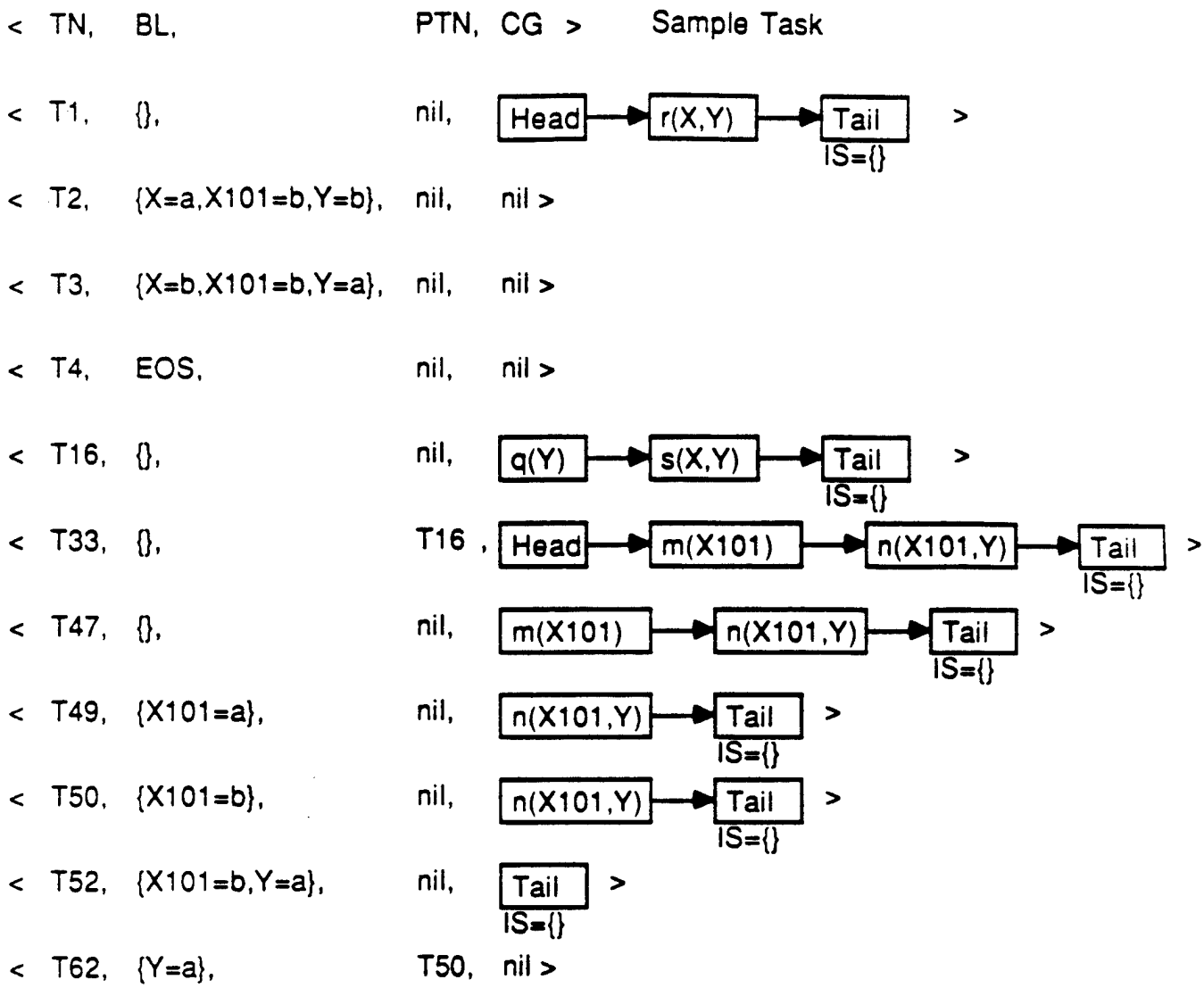


Figure 17: Some Abbreviated Task Descriptions

The *Composition* function is applied to pairs of (1) substitutions received by a process on its input tasks with (2) substitutions received from the solution of the child tasks. The composed substitutions are sent out with the outgoing tasks. For example, observe how the result of *Composition*, from its application to the substitution in input task T50 and the substitution in T62, is the substitution in output task T52.

Cartesian product of multiple input streams at a process creates new tasks with new names. T37, T38, T39, and T40 are skipped in the task numbering shown in figure 16 because they are created internally in processor P6 from the cartesian product of T18, T19 and T21, T22.

2.3.6 Remarks on Efficiency

This section contains comments on some efficiency issues related to the basic execution model.

Distributed Environments: As described earlier, substitutions of tasks on virtual input streams are retained in a process when subgoals are set up. This is the distributed environment approach. An alternative would be to send complete copies of environments to child tasks. This could be accomplished by replacing the substitution field, as it stands currently, by a stack of substitutions. However, the disadvantage with the "copying" approach is that communication costs will be higher and perhaps unacceptable. The disadvantage of distributed environments is that subsolutions must be returned to the process generating subtasks so that the *Composition* function may be applied to the input substitutions paired with the subsolution substitutions.

Number of Tasks Generated: In the example shown in section 2.3.5, 66 tasks were generated. If a sequential Prolog interpreter were used with the same database, the number of logical inferences[8] used would have been 15. One might ask if the 66/15 ratio of the number of tasks to the number of sequential logical inferences reflects on the inefficiency of the model. As it turns out, the 66/15 ratio is completely

[8] A logical inference is defined to be a successful reduction of a literal goal by either one assertion or one rule.

misleading. The number of tasks generated in a dataflow* graph is not a good indicator of the cost of the model as will be shown below.

Several simple optimizations can be used to do away with a large number of tasks entirely and many other tasks involve trivial amounts of computation. In particular, end-of-stream tasks need not be sent separately. Each end-of-stream task can be piggybacked on the last regular task sent on the stream in question. Another optimization is to replace tasks on the solution channels of Tail processes with function calls. Since each such function call involves very little work (i.e., *Composition* of two substitutions or checking whether an end-of-stream task should be sent on the output streams of the normal process), we will ignore these in the cost calculation. Also, the Head nodes merely serve as routers of data. Therefore, tasks on the task channels of Head nodes will be ignored as well in the cost calculation. Also, tasks on the input channels to Tail processes lead only to cartesian product but not to any logical inferences. We will ignore these tasks as well. The cost of cartesian product, in general, will be considered separately later in this section. After having removed all tasks from the example that are to be ignored as described above, we notice that only 10 tasks remain for which logical inferences may need to be performed. These tasks are T5, T14, T16, T18, T19, T21, T22, T47, T49, and T50.

However, to make a comparison of cost between *PM* and sequential Prolog, even this number of tasks remaining can be misleading. One should really consider the number of logical inferences that are associated with the remaining tasks. On doing the arithmetic, we find that, in this particular case, the number of logical inferences in the example is also 10. Notice that this is less than the number of logical inferences (15) in the sequential Prolog case.

The number of logical inferences in dataflow* graphs is highly dependent on the partial order that is chosen for conjunctive goals. By choosing a bad partial order, it is possible to have a higher number of logical inferences in dataflow* graphs compared to sequential Prolog. It also turns out that if no and-parallelism is exploited, and the only parallelism exploited is or-parallelism and pipelining, then the number of logical inferences is identical for both *PM* and sequential Prolog.

Also, as shown in the example, if the partial orders are chosen carefully, then the number of logical inferences can be reduced.

In addition to reducing the number of tasks, one can also reduce the number of processes. In particular, since Head processes are used as data routers only, they do not have to be created explicitly. Also, both Head and Tail processes, created when an assertion is used to reduce a literal goal, may be removed because there is an empty DAG between them.

Cost of Decomposition: Partial orders need to be generated for conjunctive goals. As mentioned before, these partial orders are of the same type used by Conery's execution model [15]. Therefore, his algorithm for partial orders can be used directly here. Also, appendix A describes another algorithm that is used in *PM* along with the associated cost.

Trade-off between Space and Time: Non-shared memory architectures (like dataflow architectures [70] and distributed systems [38]) have the property that extra space may be consumed in the attempt to reduce time of execution. This can happen if, for example, two parallel operations, O2 and O3, have a dataflow dependency on the result of an operation, O1. If memory is not shared and all three operations are on different processors, then copies of the result of O1 must be sent to the processors associated with O2 and O3. In a shared memory architecture, the processors associated with O2 and O3 could simply read a single copy of the result O1 from shared memory. The target architecture of *PM* does not have shared memory either and, therefore, shares this property.

Cost of Cartesian Product of Streams: As was pointed out before, cartesian product of streams requires extra memory compared to the sequential Prolog execution. In particular, taking the cartesian product of streams requires space equal to the sum of the lengths of the individual streams. In addition, the number of elements in the cartesian product may be equal to the product of the lengths of the streams *in the worst case*. Notice that the worst case is reached only when no composite binding leads to any inconsistent bindings.

In addition, the processing cost associated with the cartesian product function is of the order of the product of the lengths of the streams. The situation is alleviated

somewhat by the fact that the constants involved in the processing cost are fairly low. In particular, the most costly processing operation is checking to see whether a composite substitution has consistent bindings. Even more importantly, however, one can save on more costly logical inferences by using *PM* as illustrated in the example.

In conclusion, there is a cost to taking the cartesian product of streams and this could be substantial in the worst case. However, the additional parallelism gained may outweigh the cost. The total number of logical inferences may be reduced as well and this may make *PM* an attractive option for sequential processors in some cases. The example given earlier in section 2.3.5 illustrates the effect of reduction in the number of logical inferences compared to the sequential Prolog execution.

Resource Allocation: As mentioned before, the design of parallel execution models is just one of many difficult problems that must be solved to make multiprocessing a success. Resource allocation is one such problem. Notice, however, that this is not a problem restricted to this particular parallel execution model.

2.4 Extensions to Basic Model

Three extensions to the basic execution model are described in this section. The first two deal with handling storage constraints due to large databases and long streams. The third extension deals with non-ground bindings of variables.

2.4.1 Handling Storage Constraints

2.4.1.1 Large Databases

As mentioned before, the basic execution model deals with the case where all clauses that can be used to reduce any particular atomic proposition goal reside in a single processor. One can achieve this if, for example, one partitions the database on the basis of the predicate symbol of the head of the clause. Each partition is mapped onto a single processor.

Of course, it is possible that a particular partition may not fit in a single processor due to memory constraints. In addition, one may want to spread a partition over many processors to exploit the parallelism in a single backward-chaining step. The goal proposition may be unified in parallel with the heads of the relevant clauses and subtasks may be created in parallel.

The solution is to maintain a single processor as being responsible for each partition (as before). However, instead of the clauses in a partition physically residing in the responsible processor, the clauses are distributed over a certain neighborhood of the processor. One could, for example, distribute the partition over all processors within some number of message hops away from the responsible processor.

Two extra message types are required now to make the subtask creation and solution collection possible. The message types are **Do-Task** and **Done-Task**.

The *Arguments* field of the **Do-Task** message type is of the form:
<*Task-Name, Task-Description, Source-Process-Name*>

Task-Name is the name of the task that needs to be worked on. *Task-Description* is its description. *Source-Process-Name* is the identity of the process that is sending the message.

The *Arguments* field of the **Done-Task** message type is of the form:
<*Task-Name, Child-Task-Name, Destination-Process-Name, Solution*>

Task-Name is the name of the task that was originally received from the process with name *Destination-Process-Name*. *Child-Task-Name* is the name of the child-task of *Task-Name* that was created and *Solution* is the substitution that is being reported as an answer to *Child-Task-Name*. Again, end-of-stream may be indicated by an "EOS" in the *Solution* field.

The messages are used in the following way. Processes still reside at the processor responsible for the relevant partition. The relevant partition is the one that is relevant to solving the atomic proposition associated with the process. When a process receives an input-task message, it finds the incremental cartesian product as before. The new tasks are, however, not solved locally. They are sent to the neighborhood associated with the relevant partition around the processor using

Do-Task messages (i.e., each processor in the neighborhood receives a copy of the **Do-Task** message). These processors in the neighborhood create subtasks just as the single responsible processor would in the basic execution model. The difference is that solutions must be communicated back to the responsible processor using **Done-Task** messages. End-of-stream is indicated as before with the *Solution* argument set to "EOS". The difference here is that each processor in the neighborhood, including those that cannot create any solutions or subtasks, must report to the responsible processor when all subtasks have been generated using the **Done-Task** message. This is done by setting the *Child-Task-Name* argument of the message to *nil* and the *Solution* argument to "EOS". In the basic execution model, since all clauses that could be used to create a subtask were in a single responsible processor, the responsible processor knew locally when all possible subtasks had been created. The new mechanism is necessary to replace knowledge that no longer resides locally.[9]

In addition, one needs to maintain a flag at the parent-task to indicate whether all possible subtasks have been found. This flag is *false* when a task is first created using cartesian product at a process. After a *Do-Task* message is sent out to the appropriate neighborhood and after the responsible processor has received an indication from each processor that all subtasks have been generated, then the flag can be set to *true*.

Note that it is not necessary that the partition be distributed over some neighborhood of a certain processor. The distribution may be over an arbitrary set of processors. This extra flexibility may be useful for some task allocation strategies.

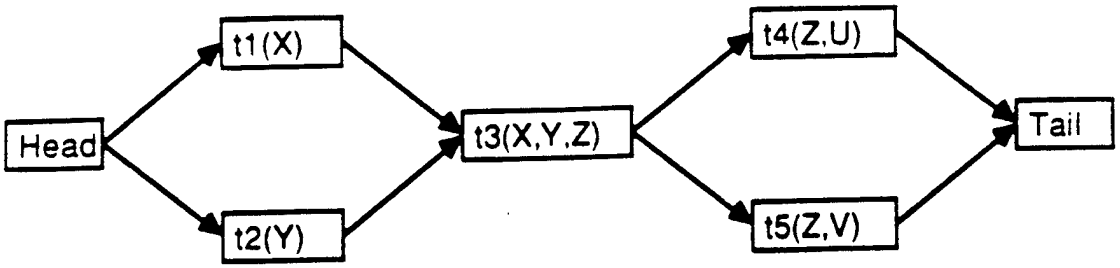
2.4.1.2 Long Streams

Processes may have multiple input streams whose cartesian product has to be computed. To create this cartesian product, essentially the process has to store complete streams until the entire cartesian product has been obtained. Since it may be hard to accurately predict the lengths of these streams ahead of time, it is possible that

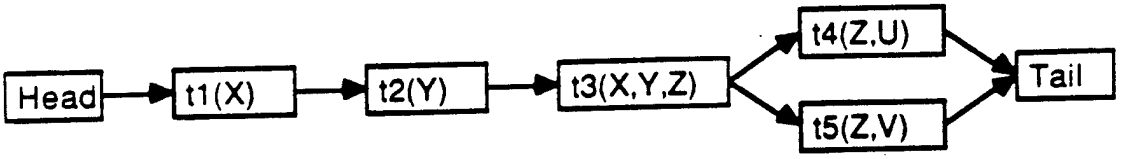
[9] A more efficient solution to propagating **Do-Task** and **Done-Task** messages to/from the neighborhood is possible but the idea here is merely to show that a *satisfactory* solution to the problem exists.

Rule h(X,Y,Z,U,V) :- t1(X), t2(Y), t3(X,Y,Z), t4(Z,U), t5(Z,V)

Goal t1(X), t2(Y), t3(X,Y,Z), t4(Z,U), t5(Z,V)



Original Dataflow* Graph



Additional Dataflow* Graph

Figure 18: Handling Long Streams

the processor responsible for the process may not have the requisite storage.

The solution is to sequentialize the dataflow* graph upstream from the process up to the Head process. As much sequentialization is done as is necessary to remove the memory problem. In the worst case, the sequentialization may lead to a linear sequence of processes requiring absolutely no cartesian products of streams. Of course, this means that no and-parallelism is exploited. Or-parallelism and pipelining will continue to be exploited as before. Figure 18 shows an example of this process. In the example, the node corresponding to t3(X,Y,Z) is the one that gets into a memory constraint situation.

Notice that the new dataflow* graph must exist independently along with the old dataflow* graph. This is necessary because tasks/solutions may still be in

the pipeline in the original dataflow* graph when the new graph is introduced. Therefore, more than one Tail process may exist for a certain task. When solutions flow out of the Tail processes, an indeterminate merge of these streams must happen. Also, an EOS is sent from the collection of Tail processes only when they have all produced an EOS.

2.4.2 Handling Non-Ground Bindings

If a process produces non-ground bindings for the atomic proposition associated with it, then some downstream processes that work in parallel may not be able to do so any more. Processes should execute in parallel only if the bindings they are expected to produce are not for any common variables. A non-ground binding from a preceding process may remove this necessary condition.

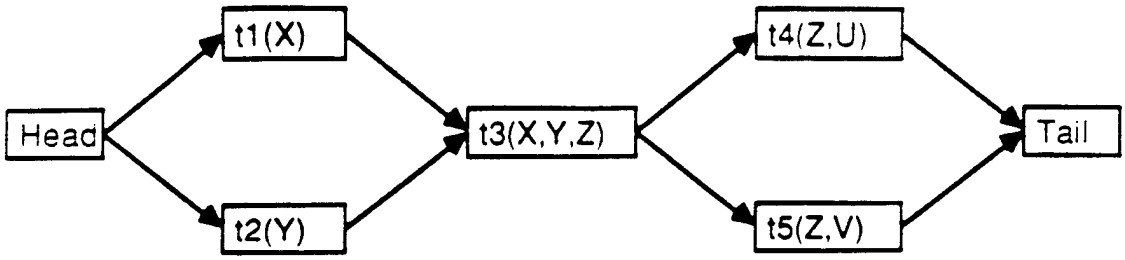
The solution is more or less complementary to the solution for the long stream problem. The dataflow* graph *downstream* from the process in question is sequentialized as much as necessary in order to avoid the problem. Figure 19 shows an example of this process. In the example, the node t3(X,Y,Z) is expected to produce a ground binding for the variable Z but does not. Similar to the long stream case, the multiple dataflow* graphs coexist independently. Multiple Tail processes are handled as before.

2.4.3 Handling Multiple Copies

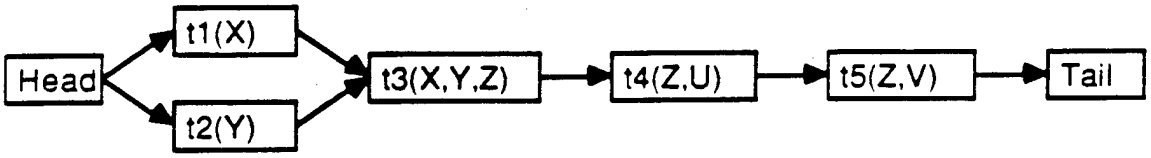
As of now, only one copy is allowed for each partition of the database. If there are goals generated in parallel that use the same partition, this restriction may lead to a bottleneck. A way out of this problem is to allow multiple copies of partitions. The solution is to decouple the functions of a normal process into two process types: CP and normal-new. The function of the CP process type is to compute cartesian products only. The normal-new process type does the rest of the computation that a normal process type did. Figure 20 shows graphically the interaction between the different process types. As indicated in the figure, two new message types are

Rule h(X,Y,Z,U,V) :- t1(X), t2(Y), t3(X,Y,Z), t4(Z,U), t5(Z,V)

Goal t1(X), t2(Y), t3(X,Y,Z), t4(Z,U), t5(Z,V)



Original Dataflow* Graph



Additional Dataflow* Graph

Figure 19: Handling Non-Ground Bindings

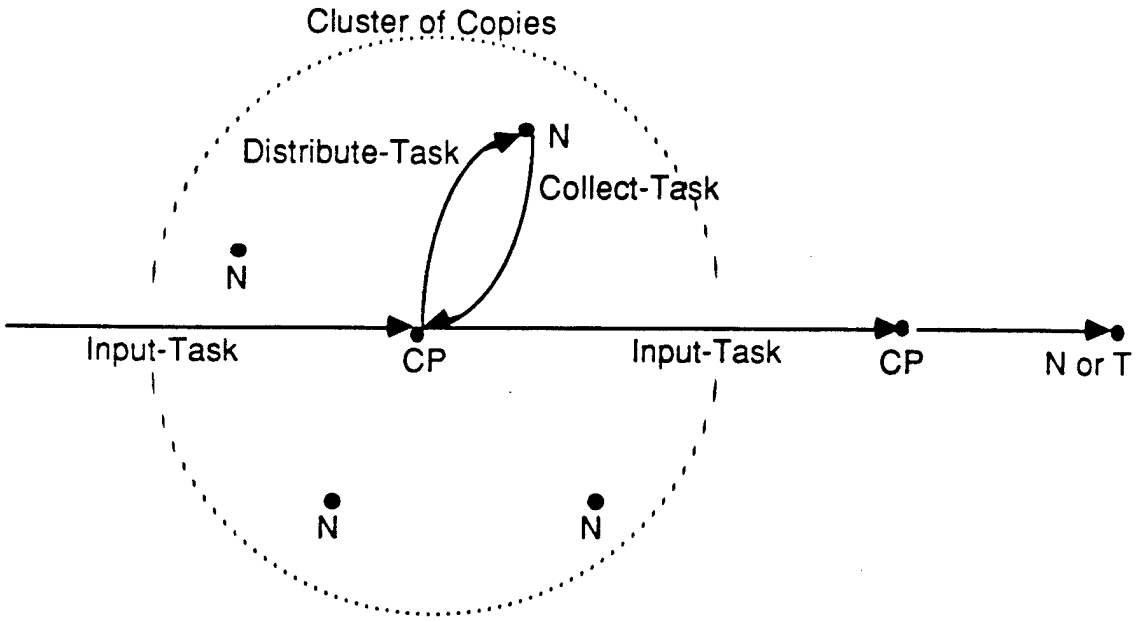


Figure 20: Handling Multiple Copies

required. These are: **Distribute-Task** and **Collect-Task**. The **Distribute-Task** message type is used to distribute computation to multiple copies of the partition and the **Collect-Task** message type is used to collect solutions from the multiple copies of the partition.

The *Arguments* field of the **Distribute-Task** message type is of the form:

<Destination-Process-Name, Task-Name, Task-Description>

The *Arguments* field of the **Collect-Task** message type is of the form:

<Destination-Process-Name, Spawned-Task-Name, Solution>

Solution is the substitution that is being reported as a solution to the task whose name is *Spawned-Task-Name*. As before, end-of-stream is indicated by an "EOS" in the *Solution* field.

Just as in the case of handling large databases, the multiple copies may be distributed to some neighborhood of a central processor or they may be in some arbitrary set of processors.

2.5 Discussion

It was mentioned earlier that side-effects are not allowed in *PM*. This is not strictly true because benign side-effects that do not affect the result of a computation but only affect the efficiency can be allowed. A side-effect of this type is caching of results. In general, the lack of general side-effects is not as severe a problem as it might seem. Many search procedures [48] do not need any side-effects. Specific applications that do not need side-effects include diagnosis [28] and test-generation [59]—both for digital hardware.

Also, it was mentioned that *PM* is designed only for non-shared memory architectures. However, it is not hard to modify *PM* to work on shared memory architectures as well. Going the reverse route (i.e., taking a shared memory algorithm and making it work on a non-shared memory architecture), is typically harder.

The rest of this section compares *PM* to related parallel execution models. The related work that is discussed in this section is work by Conery [15], Singh and Genesereth [61], Lindstrom and Panangaden [41], Ciepielewski and Haridi [12], Bic [8], Clark and Gregory [14], Shapiro [57], Borgwardt [9], and Furukawa [25].

The research presented in this chapter builds on two important ideas. One is the exploitation of and-parallelism as described by Conery in his dissertation [15]. The other is the exploitation of or-parallelism and pipelining as described by Ciepielewski and Haridi [12], Lindstrom and Panangaden [41], and Singh and Genesereth [61]. The connections of *PM* with these two sets of ideas are described below.

Conery's execution model exploited a restricted sort of and-parallelism. This restriction is exactly the one used in *PM*. A significant difference is that the backtracking control of Conery is completely abandoned. Instead, *PM* uses a dataflow solution (with the exceptions described before). One consequence is that communication is reduced because all communication associated with backtracking is absent. A second consequence is that control is more decentralized. In general, Conery's and-processes correspond to the Head/Tail process pairs used in *PM* and Conery's or-processes correspond to the normal processes in *PM*. *PM* does not have

the Head/Tail process pairs coordinate the activities of the normal processes (in between) as Conery's model had the and-processes do for the children or-processes. A third consequence is that parallelism due to pipelining comes for free in *PM*. Conery's execution model, on the other hand, sends one solution at a time along "dataflow" arcs. Further solutions are sent on the prodding of another level of control analogous to backtracking in sequential Prolog.

Haridi and Ciepielewski [12], Lindstrom and Panangaden [41], and Singh and Genesereth [61] showed how or-parallelism and pipelining could be exploited together. In these pieces of research, conjunctive goals were solved from left to right in sequence. *PM* exploits and-parallelism also by using the idea of streaming for pipelining but allows the total order of conjuncts to be changed to a partial order. Or-parallelism is exploited as before. However, the cost of exploiting the additional parallelism is that a dataflow solution (modulo indeterminate merge) has a non-dataflow feature, cartesian product of streams, added to it. Although cartesian product does require state to be maintained, the good news is that it is only local state. No global state is maintained.

Bic [8] describes another data-driven parallel execution model. However, this model only handles a restricted form of Horn clauses. Specifically, predicates must be binary, functions must be immediately evaluable during execution, and no structured terms are allowed.

Other parallel execution models have made use of programmer-supplied annotations to control the parallelism. Examples include Clark and Gregory's PARLOG [14], Shapiro's Concurrent Prolog [57], and Borgwardt's execution model [9]. *PM* differs from these execution models in that it does not use any annotations. Another difference is that none of these three execution models exploits pipelining as exploited by *PM*. Moreover, Borgwardt's execution model is restricted to shared-memory architectures. However, these execution models have been characterized by the exploitation of another form of parallelism—stream parallelism. As defined by Conery [15], this type of parallelism involves the pipelining of structured data. For example, if two functions are to be applied, in sequence, to a list of data elements, the first function may be applied to the elements of the list one by one and these

partial results may be sent to the second function as they are generated. The second function may be applied to the result elements as they are generated by the first function. Typically, this form of parallelism is not important for knowledge-based applications. For example, the diagnosis [28] and test-generation [59] applications mentioned before do not contain any exploitable parallelism of this type.

Matsumoto et al., in their backup parallelism model [25], view each node of the and-or tree as a process. Each and (or) process activates descendant or (and) processes. A descendant process starts searching for another solution right after it sends a solution to the parent process. If an additional solution is found and it is not needed by the parent, the descendant process suspends. If the process is reactivated by the parent process in the future, it immediately returns a previously found solution or continues trying to find a solution. Therefore, one level of or-parallelism is maintained throughout the tree of processes. *PM* does not restrict the level of or-parallelism.

After the work on *PM* was originally published [63,62], Li [40] came up with essentially the same idea independently for her doctoral dissertation. She calls her parallel execution model the Sync Model.

The list of parallel execution models compared to *PM* in this section is by no means exhaustive. An attempt was made, however, to cover all major categories that are relevant.

2.6 Conclusions

This chapter has described *PM*, a parallel execution model for backward-chaining deductions. The most important contribution of this chapter is that *PM* can simultaneously exploit *or-parallelism*, *and-parallelism*, and *pipelining*. This is more parallelism than is exploited by other execution models using dataflow principles, multiprocessors with no shared memory, and distributed databases. The extra parallelism can be an important advantage in a situation where large numbers of processors are available. Using dataflow principles means that synchronization overhead is minimized and the inherent parallelism can be fully exploited.

Chapter 3

Cost Function

Optimal task allocation, even for relatively simple problems, is NP-complete [43]. The approach taken in this thesis is to define a cost function that quantifies intuitive notions of undesirable allocations and yet allows for efficient computation and recomputation. This chapter describes the cost function formally and presents algorithms for its computation and recomputation. The next chapter describes the allocation algorithms that use this cost function and results obtained from an implementation of the allocator.

This chapter is organized as follows. Section 3.1 gives a formal definition of the cost function. The next two sections describe algorithms to compute this cost function.

3.1 Definition of Cost Function

3.1.1 Preliminary Definitions

The logic program is described as a 3-tuple $< F, R, G >$, where F is the set of facts (i.e., Horn clauses with exactly one positive literal and no negative literals), R is the set of rules (i.e., Horn clauses with exactly one positive literal and one or more negative literals), and G is the set of goals (i.e., conjunctions of positive literals).

Both facts and goals at compile-time may contain *unknown constants*. Unknown constants exist at compile-time only and represent constants at run-time. They are called *unknown* because their exact values at run-time are not known at compile-time. Since facts/goals with unknown constants may represent one of potentially many actual facts/goals with constants, facts/goals with unknown constants may be called *fact patterns* or *goal patterns*. For example, a fact pattern $p(uc)$, where uc is an unknown constant, may represent either of facts $p(c1)$ or $p(c2)$, where $c1$ and $c2$ are actual constants.

Fact and goal patterns may be specified with an associated number. The number represents the expected number of *instances* of those fact and goal patterns at run-time. An instance of a fact/goal pattern is a fact/goal with specific values substituted for all unknown constants in the fact/goal pattern.

$L\,|_S$, where L is a literal and S is a substitution, denotes the literal obtained by applying the substitution S to the literal L.

A *cluster of processors* is defined to be a set of processors that includes a central processor for the cluster and all processors within some specified distance away from the central processor. The size of the cluster can vary depending on the maximum distance allowed from the central processor and processors on the periphery of the cluster. Given the FAIM-1 topology as described in 1, these cluster sizes can be $3E(E-1)+1$ for positive integer E. The maximum number of processors in a cluster is restricted to be less than or equal to the maximum number of processors in the multiprocessor.

3.1.2 Assumptions

The algorithms to compute the cost function depend on the assumptions listed below:

1. **Unknown constants must represent atomic constants at run-time. They may not represent compound terms.**

This assumption is chosen so that it will be possible to estimate the amount of data (in bytes, say) that will be used to represent these constants at run-time. If they represent arbitrary structures or functionals, then it may not be possible to estimate the amount of data without additional information from the user.

One way to satisfy this assumption is to not allow any structures or functionals at all. However, one does not have to be this strict because all that is required is that unknown constants not be bound to structures or functionals. Chapter 4 (on Allocation Algorithms) contains an example that has to do with reasoning about a digital hardware device. In that example, functionals are present, yet unknown constants are always bound to atomic constants.

2. There are no recursive clauses in F ∪ R.

   With arbitrary recursive clauses, it is not possible to estimate the amount of communication or computation because the problem is equivalent to the halting problem. (It will be seen in section 3.1.3 that estimating the amount of communication and computation is necessary to compute the cost function.) However, in certain recursive cases, it may be possible to estimate the amount of communication and computation automatically. For example, if the length of a list argument gets reduced by one for every recursive call, then the recursion depth can be estimated to be the length of the list and it should be possible to estimate communication and computation. In addition, there may be other cases where some pragmas (or hints) from the user may allow a program to complete the rest of the analysis. For example, in a quick sort program, the length of the list gets reduced to half for every recursive call. Therefore, the recursion depth is $log_2 n$, where n is the length of the list.

3. Each fact in F is ground (i.e., no variables are allowed in any fact). Rules may contain variables, however.

   Again, this assumption is designed so that proper estimates may be made of the amount of communication and computation. In particular, this assumption makes it possible to know which DAG will be used for a particular set of

conjuncts in a conjunctive goal. Remember that section 2.4.2 had described how to handle non-ground bindings. This complication can be ignored when this assumption is made.

4. Equal frequency assumption: An unknown constant is equally likely to represent any known constant in the associated domain.

   For lack of any more information, this assumption seems as good as any. The question arises, however, of what to do if more precise information is given about the probability distribution of the unknown constant values. This thesis does not make a contribution here. It should be noted though that this assumption will be used later to compute the probability of two literals unifying. That computation is completely independent of other parts of this thesis. Therefore, if techniques are found for taking other probability distributions into consideration, then they can be used immediately with no change to the rest of the thesis.

5. Variable independence assumption: During unification of two literals, we assume that each distinct variable varies independently over its domain.

   Again, this assumption is made to allow computation of the probability of unification of two literals. And again, this issue is orthogonal to the rest of the thesis. Therefore, other techniques for estimating probability may be used freely.

6. Literal independence assumption: Solutions of individual conjuncts in a conjunctive goal are independent of each other.

   The same comments that applied to the two previous assumptions apply here as well.

7. Multiple copy clustering assumption: Although multiple copies of a single partition may be distributed over the set of processors in an arbitrary way, we consider the restricted case in which all processors in a cluster of processors contain a copy of the partition (and no other processors contain a copy).

This assumption is probably not going to be the best distribution of multiple copies for all applications. However, for the applications considered in this thesis, this assumption is reasonable. It will be argued in chapter 4 while discussing experimental results that this assumption is reasonable for a fairly wide class of applications—the class of applications in which there is a high degree of locality of computation. Reasoning about digital hardware seems to exhibit this locality. Reasoning about other physical artifacts may exhibit this locality as well.

The alternative of allowing arbitrary locations of copies may not be unfeasible but leads to more expensive cost computation/recomputation and allocation search algorithms. Therefore, if it is not necessary, as in the applications considered in this thesis, then it is best to use the "clustered copies" approach.

8. Multiple copy uniformity assumption: Again, in the general case, multiple goals associated with the same partition may be distributed in an arbitrary way over multiple copies of the partition. We consider the restricted case in which goals are uniformly distributed over the multiple copies. In particular, the uniform distribution is done by assigning any new goal to a random copy of the associated partition.

   The same comment that applied to the previous assumption applies here as well.

3.1.3 Cost Function

C, the cost function, takes an allocation as defined in chapter 1 and returns a non-negative real. Actually, since multiple copies are restricted to clusters as described above, an allocation can now be restated to be a many-to-one (instead of many-to-many) mapping of partitions to processors. The processor mapping of a partition is taken by convention to be the central processor in the associated cluster of copies.

I will now give some motivation for the cost function before defining it formally. Every parallel computation has an associated *parallelism profile*, where *parallelism*

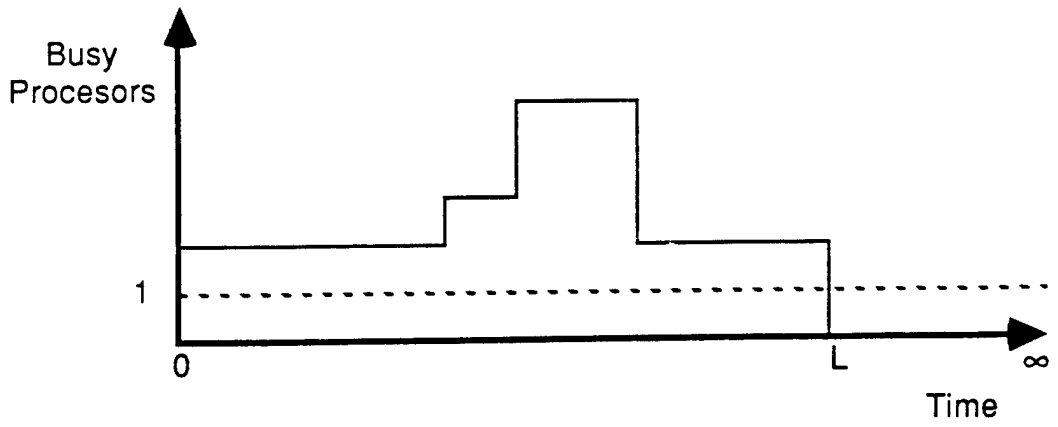


Figure 21: Parallelism Profile of a Computation

profile is defined to be the function the gives the number of busy processors versus time assuming unbounded processors and memory, and instantaneous communication. Let us say that the profile is as given in figure 21. Now, a lower bound on the completion time for the computation for any practical multiprocessor is given by L (because any practical multiprocessor will have a bounded number of processors and non-zero communication delays). If A is an allocation, a cost function C' can be defined as follows:

$$C'(A) = L + CC(A) + PMC(A)$$

where $CC(A)$ (or the *communication cost* of the allocation) is the additional delay expected due to non-zero communication delays in a practical multiprocessor and $PMC(A)$ (or *processor multiplexing cost* of the aliocation) is the additional delay expected due to sequentialization of parallel computations. Notice that L is independent of any allocation. Therefore, if the only purpose of using the cost function is to compare multiple allocations, a new cost function C can be defined as follows:

$$C(A) = CC(A) + PMC(A)$$

In general, there is a trade-off between CC and PMC. Allocating all computation to a single processor makes CC zero. However, PMC is the highest for this situation since all parallel computation needs to be sequentialized. On the other hand, if the computation is spread out among as many processors as possible (assuming for now that there is no shortage of processors), then PMC is lowest. However, CC is the highest for this situation. Finding a good allocation depends on finding a good tradeoff between CC and PMC.

Notice that no parallelism is exploited within any given partition. Therefore if the dataflow* graph is as given in figure 22, where the dashed lines enclose computation within partitions, then only communication and parallelism *across* partition boundaries make contributions to CC and PMC respectively.

3.1.4 Communication Cost Function

CC, the communication cost function, is defined to be the sum of delays of all the messages that need to be sent. This is an upper bound on the extra delay that should be expected due to non-zero communication delay. The upper bound will be reached if all the communication is on the critical path. A closer upper bound might take parallelism of communication into account and this is explored a bit in chapter 4. It turns out that the current definition of the communication cost function works quite well (as will be seen in chapter 4).

Let $delay(dt, ds)$ be the time taken for a message with data size dt to travel from a source to a destination separated by distance ds. The units for dt and ds could be bytes and hops respectively, for example. In the FAIM-1 multiprocessor, extensive simulation has shown [67] that the delay function is expected to be of the form given below.

$$delay(dt, ds) = \begin{cases} K_1 + K_2 \times dt + K_3 \times ds & \text{if } ds > 0 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where K_1,K_2, and K_3 are constants. Note that $ds = 0$ means that the source and destination processors are the same.

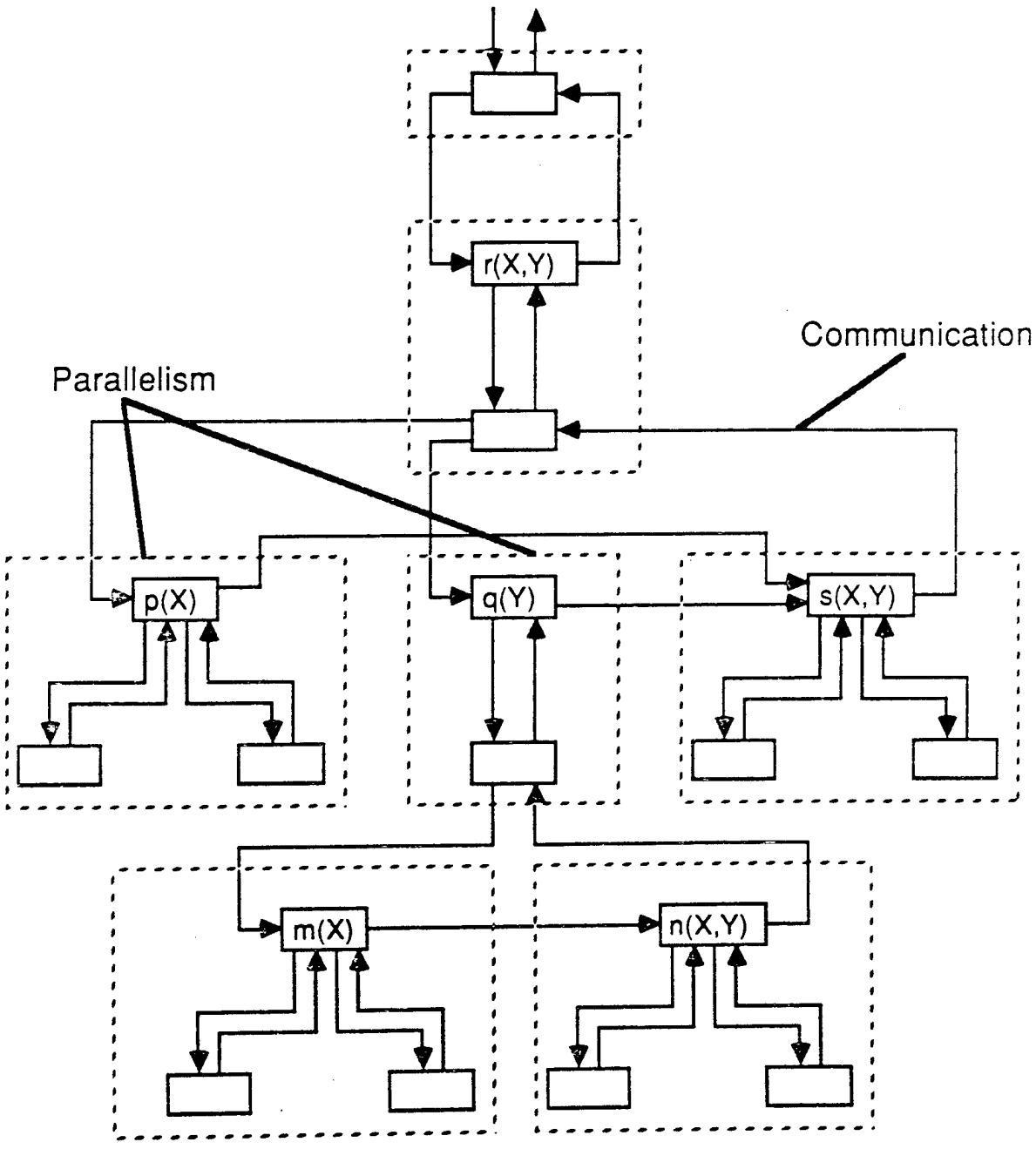


Figure 22: Partitioned Dataflow* Graph

Formally, we can say

$$CC(A) = \sum_{\forall M_j \in SM} delay(data(M_j), distance(M_j)) \quad (2)$$

where SM is the set of messages that need to be sent, and *data* and *distance* are functions that give the data size and distance (between source and destination processors) of a message.

As will be seen later in the description of the algorithm to compute communication cost, it is useful to reformulate equation 2 using equation 1 as shown below.

$$CC(A) = \sum_{\forall i} \sum_{\forall j} SD_{i,j} \quad (3)$$

where $SD_{i,j}$ is the sum of delays for all messages that need to be sent from partition i to partition j. Now, if these partitions are mapped to the same processor, we have

$$SD_{i,j} = 0 \quad (4)$$

Let us consider the other case in which the two partitions are not mapped to the same processor. Further, let the distance between the two processors be $dist(i,j)$. Now,

$$SD_{i,j} = \sum_{\forall M_l \in SMP_{i,j}} delay(data(M_l), dist(i,j))$$

where $SMP_{i,j}$ is the set of messages that need to be sent from partition i to partition j. Substituting from equation 1, we get

$$SD_{i,j} = \sum_{\forall M_l \in SMP_{i,j}} (K_1 + K_2 \times data(M_l) + K_3 \times dist(i,j))$$

Now, let the number of messages sent from partition i to partition j be $num(i,j)$ and the total amount of data in all these messages be $data(i,j)$. Substituting this into the above equation gives

$$SD_{i,j} = K_1 \times num(i,j) + K_2 \times data(i,j) + K_3 \times num(i,j) \times dist(i,j)$$

In summary, we have the following equations for communication cost:

$$CC(A) = \sum_{\forall i} \sum_{\forall j} SD_{i,j} \tag{5}$$

$$SD_{i,j} = \tag{6}$$

$$\begin{cases} K_1 \times num(i,j) + K_2 \times data(i,j) + K_3 \times num(i,j) \times dist(i,j) & \text{if } dist(i,j) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Therefore, to compute the communication cost function, it is sufficient to know the total number of messages and the total amount of data to be sent between each ordered pair of partitions. Notice on the right hand side of equation 6 that only $dist(i,j)$ is dependent on the particular allocation being considered. Therefore, if a different allocation is considered, very little recomputation needs to be done to compute $SD_{i,j}$ and in turn CC.

In case multiple copies of partitions are allowed, there will be some additional communication between the CP processes and the associated normal-new processes (see section 2.4.3). Also, if the communication is non-zero, the communication cost given above in equations 5 and 6 varies linearly with the distance. Therefore, the additional communication cost can be accounted for very easily by associating it with a distance that is the expected distance from the central processor of the partition to all other processors that contain copies of the partition. This is reasonable because the multiple copy uniformity assumption dictates that multiple copies of partitions are used equally (in a probabilistic sense).

3.1.5 Processor Multiplexing Cost Function

Informally, the processor multiplexing cost function PMC ignores all communication cost (i.e., assumes instantaneous communication) and increments cost for every instance in which two tasks could be done in parallel but are assigned to the same processor.

PMC is defined with respect to a hypothetical world and not the real world. This hypothetical world can be defined in terms of differences from the real world. There are two differences:

1. Zero communication delays

   Messages get transmitted instantaneously in the hypothetical world.

2. Infinite pool of *virtual processors* for each actual processor

   When an actual processor receives a message, it immediately assigns (with no overhead) a free virtual processor from its pool to process the message. However, the processing of each message by a virtual processor is done in the normal sequential manner.

Given this hypothetical world, it is clearly possible to have more than one task being executed at a particular actual processor at any time. Let us define the *processor-load* $pl_{i,j}(t)$ of an actual processor P_j at time t for top-level goal G_i to be the number of tasks generated from G_i being executed at P_j at time t. A particular $pl_{i,j}(t)$ may look like the curve in figure 23. Also, *excess-processor-load* is defined to be the excess over 1 of the processor load. In other words,

$$epl_{i,j}(t) = \max(0, pl_{i,j}(t) - 1)$$

In figure 23, $epl_{i,j}(t)$ is the value of $pl_{i,j}(t)$ over the $y = 1$ dashed line. Since there is only one unit of processing power available at each processor, $epl_{i,j}(t)$ represents computation that must be sequentialized. An upper bound on the additional time taken due to this sequentialization is represented by the shaded area above the $y = 1$ line. The upper bound is reached if all the computation that must be sequentialized is on the critical path of the computation. The sum of these shaded areas for all processors weighted by the top-level goals is defined to be the processor multiplexing cost. To be more precise,

$$PMC(A) = \sum_{\forall G_i \in SG} numgoal(G_i) \times \sum_{j=1}^{q} \int_0^\infty epl_{i,j}(t)dt \qquad (7)$$

where SG is the set of top-level goals, $numgoal(G_i)$ is the number associated with top-level goal G_i, there are q processors named $P_1 \ldots P_q$, and $epl_{i,j}(t)$ is the *excess-processor-load* of actual processor P_j considering only top level goal G_i.

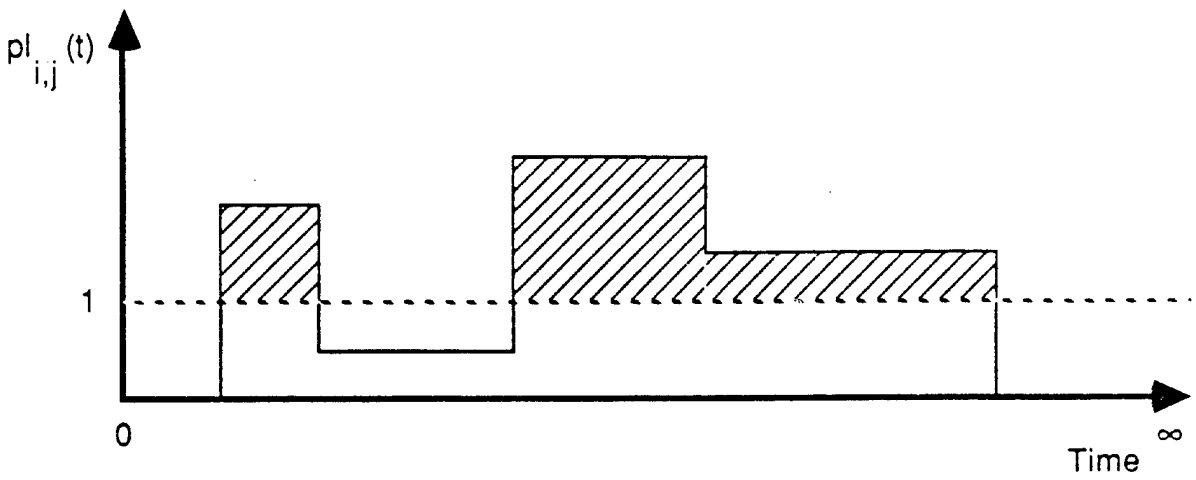


Figure 23: Processor Load Function

One way to compute the processor multiplexing cost is to first compute what is necessary for any allocation. Then, additional computation can be done to take a specific allocation into account. In particular, *processor-load* can be computed for each partition assuming it is allocated to a processor separate from any other partition. The following allocation-specific computation must be done for each top-level goal and processor. *Processor-loads* of all partitions that are allocated to a particular processor P_j for a particular top-level goal G_i must be combined to get $pl_{i,j}(t)$. The "shaded-area" computation can now be done for each processor and top level goal combination and then these can be summed up according to equation 7.

In case there are multiple copies, the *processor-load* associated with any particular partition is assumed to be equally distributed over the multiple copies of the partition in question.

3.2 Strategy for Computing Cost Function

To compute the cost function exactly requires doing the run-time computation at compile-time. Since this is clearly senseless, we restrict compile-time computation to reasoning about an abstraction of the run-time computation, an abstraction in

which specific constants of the facts in the database are ignored. The hope, of course, is that the approximation to the run-time computation is close enough to get meaningful numbers from the analysis. In addition, it is hoped that much less computation needs to be done to reason with the abstraction instead of the real run-time computation.

Figure 24 illustrates how *fact patterns* with unknown constants replace facts with actual constants in the database. Symbols beginning with "uc" represent unknown constants. The crossed out facts are the ones in the original database. Figure 25 illustrates that using *fact patterns* reduces the number of logical inferences. Logical inferences enclosed in thick ovals may be collapsed into one logical inference at compile-time. In the best case, the number of logical inferences may be reduced by an exponential factor. Figure 26 shows a conjunctive goal with 3 conjuncts. If there are n facts with the a predicate, n^2 facts with the b predicate, and n^3 facts with the c predicate, then the number of logical inferences at run-time is $(n + n^2 + n^3)$ or $O(n^3)$. In general, for m conjuncts, the number of logical inferences would be $O(n^m)$. However, if the facts of each predicate get represented by a single compile-time fact, then the number of logical inferences at compile-time is only 3. In general, for m conjuncts, the number of compile-time inferences is only $O(m)$. Therefore, the number of logical inferences is reduced by an exponential factor from $O(n^m)$ to $O(m)$.

One effect of using unknown constants is that unification is now a probabilistic process. It does not just succeed or fail; it succeeds with some probability. This will be discussed in more detail in section 3.3.

Another computation-saving technique used in the cost computation procedures is to separate out the allocation-independent computation from the allocation-dependent computation. The allocation-independent computation needs to be performed only once for the application. Only the allocation-dependent computation needs to be performed when a specific allocation is being considered. In addition, when an allocation is changed slightly, even the allocation-specific computation need not be performed from scratch. Useful state can be saved between recomputations and this can lead to significant savings.

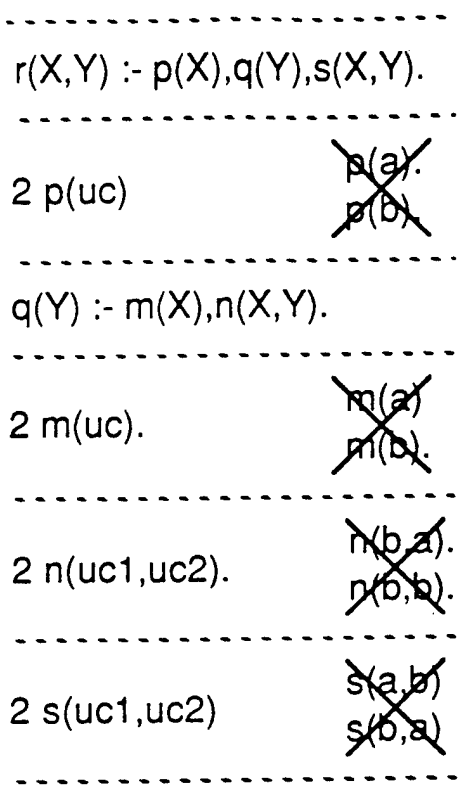


Figure 24: Compile-time Database

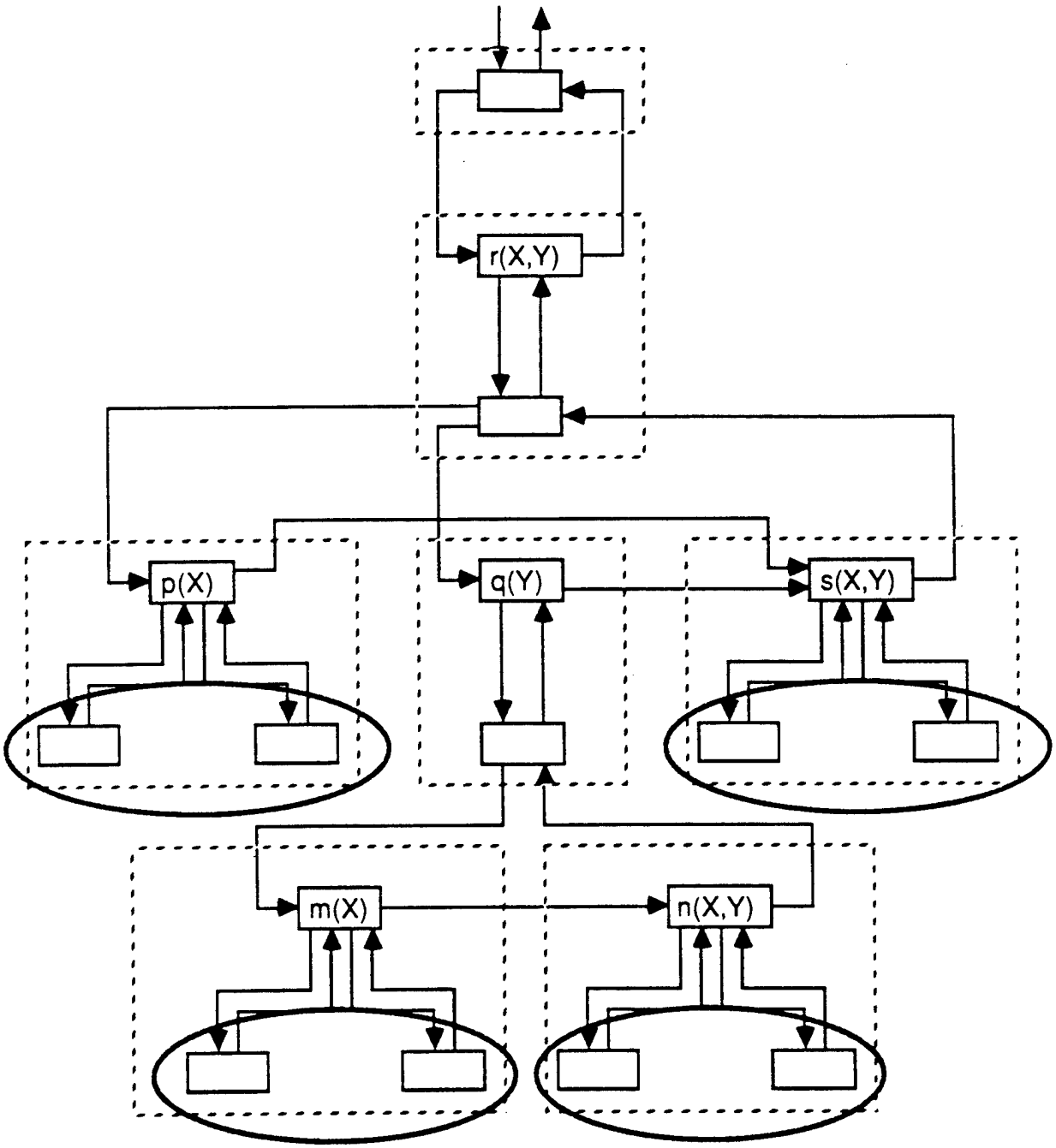


Figure 25: Compile-time Computation

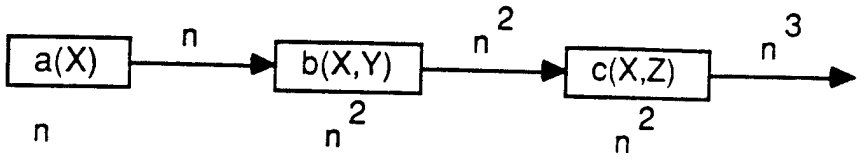


Figure 26: Exponential Savings at Compile-time

In the case of communication cost computation, the number of messages and the amount of data between each pair of partitions is independent of the allocation. Only, the distance between partitions is dependent on the allocation being considered.

In the case of processor multiplexing cost computation, the *processor-load* functions associated with each partition (assuming that they are allocated to separate processors) are independent of the allocation. Combining *processor-loads* of different partitions does depend on the allocation. However, useful state can be kept between recomputations to save on computational effort (as will be seen later in section 3.4).

An alternative to this approach of estimating the amounts of communication and *processor-loads* at compile-time is to gather information from one or more runs of the application and collect this information for use by the cost computation procedures. One can also think of hybrid approaches in which compile-time estimates may be modified (if necessary) by data collected at run-time. The advantage with using run-time data is that one does not depend on assumptions that may not be totally accurate (required to do compile-time estimation and listed in section 3.1.2). However, the disadvantage is that the estimates may get unduly influenced by the last run or last several runs. Also, making several runs of the application can be much more expensive (depending on the number of runs) than making one run using unknown constants.

3.3 Communication Cost Computation

The algorithms described in this section are for computing the communication cost for a single top-level goal. If there are multiple top-level goals, then the algorithms need to be repeated for each top-level goal and the costs summed. Also, if a certain top-level goal is repeated multiple times, the communication cost is computed for a single instance and the communication cost for the multiple instances is computed by multiplying the single instance cost by the repetition factor.

The computation of *communication cost* is done by two algorithms. The first algorithm is called the *Communication Estimation* algorithm. This algorithm performs an abstract simulation of *PM*. A side-effect of the simulation is the estimation of the amount of communication (in total bytes and number of messages) between every pair of partitions. The second algorithm is called the *Communication Cost Computation* algorithm. This algorithm takes the output of the *Communication Estimation* algorithm and an allocation and computes the *communication cost*.

The *Communication Estimation* algorithm is based on the idea of simulating (at compile-time) a backward-chaining deduction using *PM* as the execution model. The difference from the actual run-time computation is that the compile-time simulation is less detailed and, therefore, takes less time than the run-time computation. Probabilistic analysis replaces some of the detailed computation and most of the description of the algorithm focuses on this analysis.

The organization of this section is as follows. Subsection 3.3.1 gives the specifications of the communication estimation algorithm. The next three subsections lay down the basis of the probabilistic analysis. Subsection 3.3.2 describes how goals may be viewed as probabilistic filters over their solution domains. Subsection 3.3.3 describes how the probability of unification of two literals may be estimated when the exact constants in the literals are not known at compile-time. Subsection 3.3.4 describes how run-time messages must be augmented to make them suitable for the probabilistic analysis. The next two subsections describe two variants of the *Communication Estimation* algorithm. Subsection 3.3.5 describes the simpler variant that does not deal with duplicate solutions while the next two subsections

show how duplicate solutions can be handled.

The *Communication Cost Computation* algorithm is trivial compared to the *Communication Estimation* algorithm. After the *Communication Estimation* algorithm has produced an estimate of the amount of communication between every ordered pair of partitions, the communication cost algorithm simply uses this information and equations 5 and 6 to compute the communication cost. Since the algorithm is so simple, it will not be described in any more detail.

Finally, subsection 3.3.8 discusses the complexity of both the *Communication Estimation* algorithms and the *Communication Cost Computation* algorithm.

3.3.1 Specification of Communication Estimation Algorithm

Inputs

1. F: a set of fact patterns.
2. R: a set of rules.
3. G: a set of goal patterns.
4. P: a set of subsets of $R \cup F$ that are mutually exclusive and exhaustive. Each member of P is called a *partition*. Remember that a constraint of PM is that all clauses that may be applied to reducing any particular literal subgoal generated during the backward-chaining deduction should be included in precisely one partition. Note that we are talking about a single logical inference here, not a goal reduction involving an arbitrary number of logical inferences. As an example, there could be one member of P for each set of facts and rules with a different predicate.
5. *domsize*: a two argument function that takes a predicate name and a number specifying a field and returns the associated domain size. For example, if $parent(X, Y)$ indicates that X is a parent of Y, then $domsize(parent, 1) = 2$ since every person has two parents. Also, if the average number of children in a family is 3, we might say that $domsize(parent, 2) = 3$.

3.3 Communication Cost Computation

The algorithms described in this section are for computing the communication cost for a single top-level goal. If there are multiple top-level goals, then the algorithms need to be repeated for each top-level goal and the costs summed. Also, if a certain top-level goal is repeated multiple times, the communication cost is computed for a single instance and the communication cost for the multiple instances is computed by multiplying the single instance cost by the repetition factor.

The computation of *communication cost* is done by two algorithms. The first algorithm is called the *Communication Estimation* algorithm. This algorithm performs an abstract simulation of *PM*. A side-effect of the simulation is the estimation of the amount of communication (in total bytes and number of messages) between every pair of partitions. The second algorithm is called the *Communication Cost Computation* algorithm. This algorithm takes the output of the *Communication Estimation* algorithm and an allocation and computes the *communication cost*.

The *Communication Estimation* algorithm is based on the idea of simulating (at compile-time) a backward-chaining deduction using *PM* as the execution model. The difference from the actual run-time computation is that the compile-time simulation is less detailed and, therefore, takes less time than the run-time computation. Probabilistic analysis replaces some of the detailed computation and most of the description of the algorithm focuses on this analysis.

The organization of this section is as follows. Subsection 3.3.1 gives the specifications of the communication estimation algorithm. The next three subsections lay down the basis of the probabilistic analysis. Subsection 3.3.2 describes how goals may be viewed as probabilistic filters over their solution domains. Subsection 3.3.3 describes how the probability of unification of two literals may be estimated when the exact constants in the literals are not known at compile-time. Subsection 3.3.4 describes how run-time messages must be augmented to make them suitable for the probabilistic analysis. The next two subsections describe two variants of the *Communication Estimation* algorithm. Subsection 3.3.5 describes the simpler variant that does not deal with duplicate solutions while the next two subsections

Output

- C: a function that takes two partitions P_1 and P_2 and returns a tuple of the form $< data, number >$ where *data* is the amount of data (in bytes) and *number* is the number of messages sent from partition P_1 to partition P_2. *data* and *number* are expected values in a probabilistic sense.

3.3.2 Goals as Filters

Each goal, be it a literal or a conjunction of literals, can be characterized as a *filter* over its solution domain. *Filter probability* is defined to be the probability that a random member of the set of possible solutions is a member of the set of actual solutions.

The cardinality of the domain of possible solutions (of a literal goal or a conjunctive goal), N_p, is given by the following equation:

$$N_p = \prod_{v_i \in V} d(v_i) \tag{8}$$

where V is the set of variables in the goal and $d(X)$ is the size of the domain of variable X. This formula assumes that if the same variable occurs more than once in a single conjunct or in more than one conjunct in a conjunctive goal, then its domain is the same for each occurrence.

If the number of actual solutions is N_a, then the filter probability, FP, is given by

$$FP = \frac{N_a}{N_p} \tag{9}$$

By using the literal independence assumption, it follows directly that the filter of a conjunctive goal is equal to the product of the filters of the individual conjuncts. In other words, the expected number of solutions N to a conjunctive goal

$$C = \{C_1, C_2, \ldots, C_n\}$$

is given by

$$N = N_p \times \prod_{i=1}^{n} FP(C_i) \tag{10}$$

where N_p is the number of possible solutions and $FP(C_i)$ is the filter probability of conjunct C_i. Plugging in the value of N_p from equation 8, we get

$$N = \prod_{v_i \in V} d(v_i) \times \prod_{i=1}^{n} FP(C_i) \tag{11}$$

where V and d are as defined before. This is an important equation because it makes the *Communication Estimation* algorithm particularly simple as will be seen later.

As an example of the application of this equation, see figure 27. A 3-conjunct goal has to be solved with the database and domain sizes as given. In this example,

$$N_p = d(X) \times d(Y) \times d(Z) = 12 \times 4 \times 2 = 96$$

Also, the filter probabilities of the three conjuncts can be computed as follows. The filter probability of "a(X)" is its actual number of solutions (= 6) divided by its potential number of solutions (= $d(X) = 12$), which is 0.5. The filter probability of "b(X,Y)" is its actual number of solutions (= 24) divided by it potential number of solutions (= $d(X) \times d(Y) = 12 \times 4 = 48$), which is 0.5. The filter probability of "c(X,Z)" is its actual number of solutions (= 8) divided by its potential number of solutions (= $d(X) \times d(Z) = 12 \times 2 = 24$), which is 0.33. Therefore, using equation 11, we get

$$\begin{aligned} N &= \prod_{v_i \in V} d(v_i) \times \prod_{i=1}^{n} FP(C_i) \\ &= 96 \times \prod_{i=1}^{n} FP(C_i) \\ &= 96 \times 0.5 \times 0.5 \times 0.33 = 8 \end{aligned}$$

Now, we carry this analysis a step further. Each conjunct in a conjunctive goal may actually be reduced by more than one rule or by more than one fact. Therefore, more than one path of reasoning may lead to actual solutions for the conjunct. Each such path of reasoning, or a set of such paths of reasoning considered

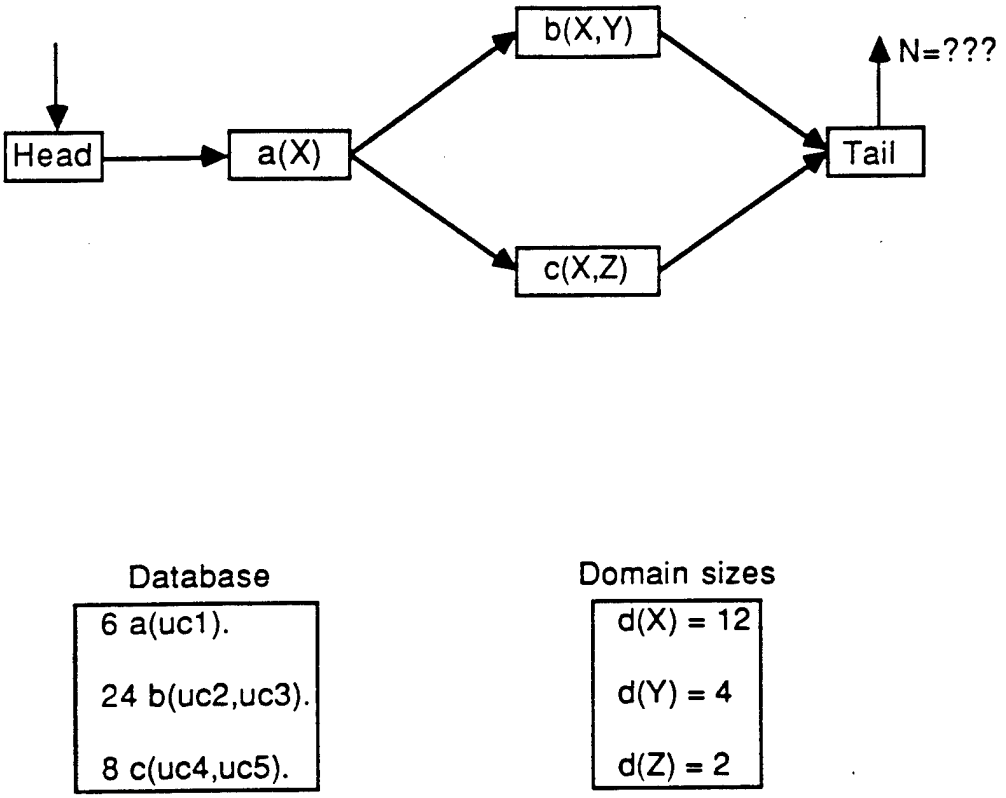


Figure 27: Predicting Communication

together, may lead to a particular set of actual solutions. This particular set of actual solutions will be a subset of the complete set of actual solutions but can be characterized as a filter nonetheless. Of course, the filter probability associated with a conjunct, considering only a subset of the paths of reasoning, will be less than or equal to the filter probability associated with the conjunct when all the paths of reasoning are considered. Due to the literal independence assumption, this filter probability associated with a conjunctive goal will be a product of the filter probabilities associated with the individual conjuncts for the same subset of the actual solutions.

3.3.3 Probability of Unification

This section describes how one can compute the probability of unification of two literals. Each literal can contain unknown constants. During unification, variables, constants and unknown constants may be unified against each other. A valid unifying substitution may contain bindings of: (1) variables to either variables, unknown constants, or constants, and/or (2) unknown constants to either unknown constants or constants. Of course, constants to be unified must match exactly.

Table 1 gives the probabilities of these unifications. In the table, $d(uci)$, where uci is an unknown constant, refers to the domain size (given by the function $domsize$) of the field of the relation that uci is associated with. The probability of unification of the two literals is simply the product of the probabilities of unifications of the type given in the table. Taking a product is justified by the argument independence assumption. Similar probabilistic analysis has been used before by Treitel in his work on selecting the optimal mix of forward and backward inference for a sequential processor [69].

As an example, consider the unification of the two literals $a(uc1, uc2, uc3)$ and $a(X, X, c1)$. In this case, $uc1$ and $uc2$ must be unifiable and the probability of this can be found from table 1 to be $\frac{1}{d(uc1)} = \frac{1}{d(uc2)}$. Notice that the domain sizes of the two unknown constants have been assumed to be equal. Also, $uc3$ must be unifiable with $c1$ and the probability of this can be found from the table to be $\frac{1}{d(uc3)}$. The

	V2	c2	uc2
V1	1	1	1
c1	1	1 (if c1=c2) 0 (if c1≠c2)	$\frac{1}{d(uc2)}$
uc1	1	$\frac{1}{d(uc1)}$	$\frac{1}{d(uc1)}$ = $\frac{1}{d(uc2)}$

Table 1: Probabilities of Unification

probability of unification of the two literals is the product of these two probabilities.

As another example, consider figure 27 again. Remember that back in chapter 2, we had proved a theorem that stated that the set of solutions produced by *PM* is equal to the set of solutions produced by a Prolog interpreter. Of course, this theorem also implies that the cardinalities of the sets of solutions must be equal in the two cases. In section 3.3.2, we saw that applying equation 11 had given the number of solutions of the conjunctive goal in figure 27 to be 8. Now, we can get the number of solutions by using a total order of conjuncts as in a Prolog interpreter and using table 1 directly and show that we get the same number of solutions. In particular, a Prolog interpreter might use a total order like the one shown in figure 28. The number of solutions of the first conjunct will be 6 because there are 6 "a" facts in the database and the variable X in the goal unifies with probability 1 with uc1 in the facts. Next, the variable X in the second conjunct gets bound to an unknown constant. The probability of unification of the "b" conjunct with the "b" facts in the database is the inverse of the domain size of the first field of the "b" relation ($= d(X) = 12$ in figure 28) since an unknown constant is getting

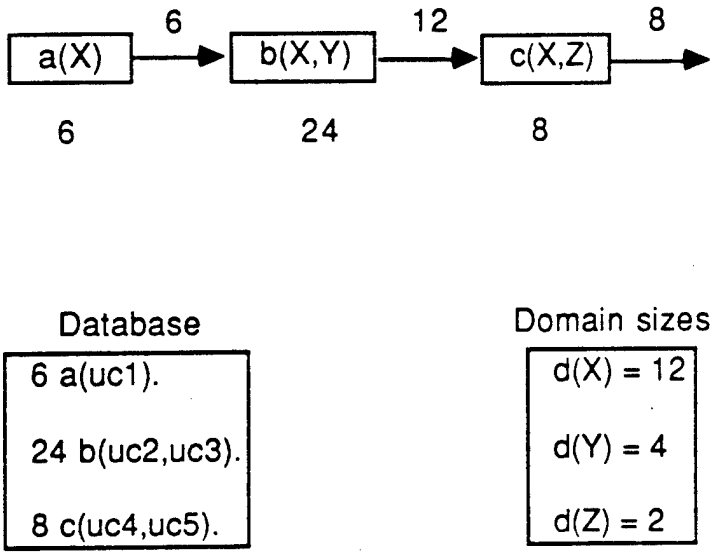


Figure 28: Estimating Number of Solutions for Prolog

unified with another unknown constant. Since there are 6 "b" goals, 24 "b" facts in the database, and the probability of unification is $\frac{1}{12}$, the expected number of solutions of the first two conjuncts is $6 \times 24 \times \frac{1}{12} = 12$. Next, variable X in the "c" conjunct gets bound to an unknown constant. The probability of unification of the "c" goals with the "c" facts is the inverse of the domain size of the first field of the "c" relation ($= d(X) = 12$) since an unknown constant must be unified with another unknown constant. Since there are 12 "c" goals, 8 "c" facts in the database, and the probability of unification is $\frac{1}{12}$, the expected number of solutions of all three conjuncts together is $12 \times 8 \times \frac{1}{12} = 8$. This is the same as the number obtained by applying equation 11. This technique of finding the expected number of solutions of a set of conjuncts by mapping it back repeatedly to the Prolog case can lead to very inelegant and inefficient algorithms. Using equation 11 directly turns out to be much simpler (as will be seen later in section 3.3.5).

3.3.4 Compile-time Messages

As mentioned before, the communication estimation algorithm is based on the idea of simulating (at compile-time) a backward-chaining deduction using *PM* as the execution model. The difference from run-time deduction is that unknown constants may be used at compile-time. For now, assume that a message in *PM* contains substitutions only. The initial behavioral description of *PM* made the same simplification. However, at compile-time, a message contains some additional information.

First, a substitution with unknown constants represents an equivalent class of actual substitutions with actual constants only. Each member of the equivalent class is obtained by giving each unknown constant a value in its domain. A compile-time message is associated with a number called the *number of substitutions*. This number represents how many instances of the compile-time message's equivalent class are expected (in the probabilistic sense) to be sent on the associated channel at run-time. Note that at run-time each message is treated completely independently. For example, when a new conjunct graph is created to solve a subgoal at run-time, a single message is sent to the *head process* of the new conjunct graph along its *task channel*. At compile-time, if the same message has an associated *number of substitutions* of N, then N separate conjunct graphs would actually be generated at run-time, each with its own message on the task channel to the head process.

The advantage of using the *unknown constant* abstraction is that it allows the algorithm to estimate communication cost without doing the entire deduction itself. Also, if the goal for the entire deduction is specified using unknown constants, the expected communication costs for the entire class of goals represented is computed in one pass. In contrast, the run-time execution model can only handle one specific goal at a time.

Second, a compile-time message is associated with a *filter set*. A filter set is a set of 2-tuples. There is one such 2-tuple for each literal that has been processed so far in the conjunct graph. Each tuple contains: (1) a number indicating the position (leftmost being 1) of the literal in the antecedents of the rule that generated the associated conjunct graph and (2) the filter probability for that literal that led to the set of substitutions described by the compile-time message.

In addition, each message on each channel in a conjunct graph contains the *initial number of substitutions* for the conjunct graph. The *initial number of substitutions* is the number of substitutions sent on the task channel of the head process to the *complete conjunct graph.* A complete conjunct graph is defined to mean a two-terminal DAG of processes that includes a matching pair of *Head* and *Tail* processes and all the *normal* processes in between.

Considering everything, a compile-time message is a 4-tuple:

$$< N, S, NI, FS >$$

where N is the *number of substitutions*, S is the substitution, NI is the *initial number of substitutions* and FS is the filter set.[1]

Each compile-time message is associated with a particular channel in the dataflow* graph during simulated deduction. The source node and the destination node of the channel are associated with one database partition each. The compile-time message contributes to the amount of communication between this pair of database partitions. The amount of data is

$$data(S) \times N$$

where data(S) is the amount of data (in bytes, say) that will be contained in the substitution at run-time that S represents. (S itself may contain unknown constants each of which represents a known atomic constant at run-time.) The number of messages to be sent is N, the *number of substitutions.* The total amount of communication between a pair of partitions is the sum of contributions from each message.

Section 3.3.5 describes the algorithm to compute the amount of communication between each pair of partitions for a single goal. If multiple goals are given, the algorithm must be repeated for each goal and the amounts of communication added up. This algorithm follows quite naturally from the ideas in sections 3.3.2, 3.3.3, and 3.3.4. It may be skipped without loss of continuity in the thesis. The interested reader can return to this section later for more detail.

[1] A message at run-time is a substitution at this level of detail.

3.3.5 Communication Estimation Algorithm (No Duplicates)

This section presents the behavioral description of the simulated parallel execution model. The description is similar in nature to the behavioral description of *PM* contained in section 2.3.2. The only difference is that the behavioral description of *PM* dealt with actual messages whereas simulated *PM* deals with compile-time messages. As explained before, the set of compile-time messages that is generated during simulated deduction contains sufficient information to compute the amount of expected communication between each pair of partitions.

The description is divided into four parts as before: (1) *Sim-CP*—the analog of the *CP* function of *PM*, (2) the response of a normal process to each compile-time message on its virtual input channel, (3) the response of a tail process to each compile-time message on its virtual input channel, and (4) the response of a normal process to each compile-time message on each of its subsolution channels.

A running example to make the explanations clearer is shown in figure 29. This is the same example as the one that was considered in chapter 2 except that unknown constants have been used for the facts. Also, the NF numbers to the right of the facts indicate how many facts of that pattern are present in the database. The dataflow* graph for the simulated deduction is shown in figure 30. Each compile-time message on each channel is shown in the figure. Only one compile-time message is sent on each channel for this example. This is not true for all cases. We will assume for this example that the cardinality of the domain of each variable and unknown constant is 2.

3.3.5.1 Analog of the *CP* function

As described in chapter 2, the function *CP* takes n sets of substitutions—a set for each input channel of a normal process—and returns a single set of substitutions. The output set of substitutions is the one carried on the hypothetical virtual input channel of the normal process in question. *CP* considers the cartesian product of the input sets and rejects all inconsistent composite substitutions using an auxiliary function called *Merge*.

```
r(X,Y) :- p(X), q(Y), s(X,Y)
p(uc1)                NF=2
q(Y) :- m(X), n(X,Y)
m(uc2)                NF=2
n(uc3,uc4)            NF=2
s(uc5,uc6)            NF=2

r(X,Y) is the top level goal
```

Figure 29: Example Database

Sim-CP, the simulated deduction version of *CP*, considers the cartesian product of sets of compile-time messages and rejects composite messages that contain inconsistent substitutions. The point of departure from *CP* is *Sim-Merge*, the simulated deduction version of *Merge*. *Sim-Merge* must determine consistency of substitutions as before. However, in addition to that, it must compute the other fields of a compile-time message. In particular, these fields are *number of substitutions*, *initial number of substitutions* and *filter set*.

Let there be n input channels for the normal process in question. *Sim-Merge* takes one compile-time message from each channel and either returns a compile-time message or $\perp$—a special element. The special element is used to indicate inconsistent input substitutions just as *Merge* did. Let the compile-time message on the i^{th} channel be

$$< N_i, S_i, NI, FS_i >$$

In case the input compile-time messages contain inconsistent substitutions, then the output of *Sim-Merge* is $\perp$. Substitutions are inconsistent if the same variable is bound to different known constants. Variables bound to different unknown constants are not inconsistent.

If the output is not $\perp$, it is a compile-time message

$$< N_o, S_o, NI, FS_o >$$

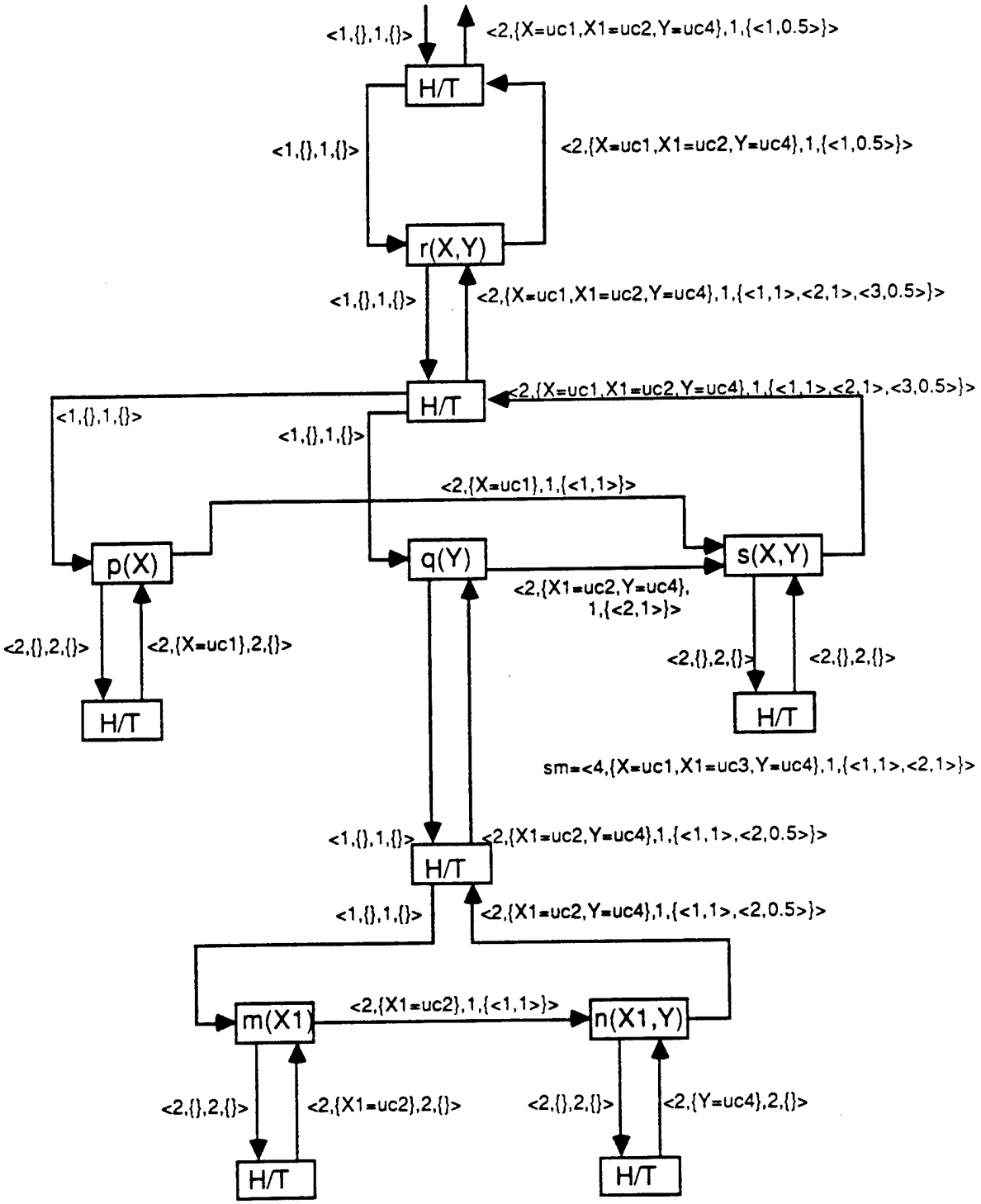


Figure 30: Dataflow* Graph for Simulated Deduction

Since a filter set contains filter tuples for each ancestor conjunct, we have

$$FS_o = \bigcup_{i=1}^{n} FS_i \tag{12}$$

Union removes duplicate filter tuples. Duplicates can arise because the same conjunct may be an ancestor from more than one input channel.

Since we are presumably considering one conjunct graph (that may be instantiated a number of times at run-time), all messages in that conjunct graph have the same *initial number of substitutions.*

Equation 11 (in section 3.3.2) showed how the expected number of solutions for a set of conjuncts could be computed. Notice that each compile-time message in the set returned by *Sim-CP* is a solution of the set of conjuncts associated with the ancestor processes of the normal process being considered. Therefore, for each initial substitution for the conjunct graph, the expected number of solutions of the ancestor conjuncts, N_{int}, that will be generated by *Sim-Merge* is given by the equation:

$$N_{int} = \prod_{v_i \in V} d(v_i) \times \prod_{FP_i \in FPS_o} FP_i \tag{13}$$

where V is the union of the sets of bound variables in the substitutions of the input messages and FPS_o is the set of filter probabilities contained in the filter tuples belonging to FS_o. For example, if

$$FS_o = \{< 1, .5 >, < 2, .75 >\}$$

where $< 1, .5 >$ indicates that the filter probability of the first literal is .5, then

$$FPS_o = \{.5, .75\}$$

Moreover, the total expected number of solutions, N_o, will be given by the equation:

$$N_o = NI \times N_{int} \tag{14}$$

Plugging equation 13 into equation 14, we get

$$N_o = NI \times \prod_{v_i \in V} d(v_i) \times \prod_{FP_i \in FPS} FP_i \tag{15}$$

Figure 30 shows this computation for the "s(X,Y)" box. The input compile-time messages are

$$< 2, \{X = uc1\}, 1, < 1,1 > >$$

and

$$< 2, \{X1 = uc3, Y = uc4\}, 1, \{< 2,1 >\} >$$

Equation 12 is used to compute FS_o.

$$FS_o = \bigcup_{i=1}^{n} FS_i = \{< 1,1 >, < 2,1 >\}$$

Equation 15 is used to compute N_o.

$$N_o = NI \times \prod_{v_i \in V} d(v_i) \times \prod_{FP_i \in FPS} FP_i = d(X) \times d(Y) \times 1 \times 1 = 2 \times 2 = 4$$

Since the substitutions are consistent, S_o is obtained by taking the union of the two input substitutions.

$$S_o = \{X = uc1, X1 = uc3, Y = uc4\}$$

3.3.5.2 Response of Normal Process to Compile-time Messages on Virtual Input Channel

In real *PM*, each message on a virtual input channel contains a substitution and this substitution applied to the literal associated with the normal process represents a goal to solve. Rules and facts associated with the normal process are applied to the goal in an attempt to reduce or solve it. The rest of this section describes the behavior of the process for one of these applicable rules/facts. The same behavior is repeated for each rule/fact.

Let the compile-time message on the virtual input channel be

$$< N_i, S_i, NI_i, FS_i >$$

and the compile-time message on the subtask channel be

$$< N_o, S_o, NI_o, FS_o >$$

The total number of actual messages represented by the input compile-time message is N_i. Some of the associated goals will unify with the literal representing the head of the rule or the fact. The probability of unification can be computed as shown in section 3.3.3. Let PU be this probability. The number of successful unifications, N_o, in case a rule is used is given by the equation:

$$N_o = N_i \times PU \tag{16}$$

In case we are dealing with a fact (as opposed to a rule) and the number associated with the fact is NF, then the number of successful unifications, N_o, is given by the equation:

$$N_o = N_i \times PU \times NF \tag{17}$$

Also, NI_o will be equal to N_o. $S_o = \{\}$ and $FS_o = \{\}$ because no conjunct in the new conjunct graph will have been been solved as yet. The invocation substitution is associated with the tail process of the conjunct graph as described in chapter 2.

The head process of the new conjunct graph passes this message unchanged to each of its output channels.

As an example, look in figure 30 at the response of the "s(X,Y)" process to the compile-time message on its virtual input channel. This message is

$$< 4, \{X = uc1, X1 = uc3, Y = uc4\}, 1, \{< 1,1 >, < 2,1 >\} >$$

Since the domain of each variable X and Y is 2, the probability of unification, PU, of the goal "s(uc1,uc4)" with the fact "s(uc5,uc6)" is

$$PU = \frac{1}{2} \times \frac{1}{2} = 0.25$$

Therefore, using equation 17

$$N_o = N_i \times PU \times NF = 4 \times 0.25 \times 2 = 2$$

NI_o is equal to N_o.

$$S_o = \{\}$$

$$FS_o = \{\}$$

3.3.5.3 Response of Tail Process to Compile-time Messages on Virtual Input Channel

Let the compile-time message on the input channel be

$$< N_i, S_i, NI_i, FS_i >$$

and the compile-time message on the task channel be

$$< N_o, S_o, NI_o, FS_o >$$

Since the tail process does not solve any goal as such, $N_o = N_i$, $NI_o = NI_i$, and $FS_o = FS_i$. The only difference is that

$$S_o = Composition(IS, S_i)$$

where IS is the *invocation substitution.*[2]

As an example, look in figure 30 at the response of the tail process below "n(X1,Y)" to the message on its input channel. The message is

$$< 2, \{\}, 2, \{\} >$$

Notice that, in this case, since there is only one input to the tail process, the virtual input channel is the same as the input channel. The invocation substitution is $\{Y = uc4\}$. Therefore,

$$S_o = Composition(\{Y = uc4\}, \{\}) = \{Y = uc4\}$$

The rest of the components of the message on the solution channel are the same as the ones for the message on the input channel.

3.3.5.4 Response of Normal Process to Compile-time Messages on Subsolution Channels

In this case, the computation depends on the compile-time message (on the virtual input channel to the normal process) that is associated with the solution being reported (on the subsolution channel). Let this compile-time message be

[2]See chapter 2 for a description of *invocation substitution.*

$$< N_1, S_1, NI_1, FS_1 >$$

Also, let the compile-time message on the subsolution channel be

$$< N_2, S_2, NI_2, FS_2 >$$

and the compile-time message on the output channel from the normal process be

$$< N_3, S_3, NI_3, FS_3 >$$

First, since the message on the output channel is associated with the same conjunct graph, we have

$$NI_3 = NI_1 \tag{18}$$

Second, just as in real PM,

$$S_3 = Composition(S_1, S_2) \tag{19}$$

Since the total number of messages on the subsolution channel must be the same as the total number of messages on the output channel of the normal process,

$$N_3 = N_2 \tag{20}$$

For each real message on the virtual input channel, the cardinality of the domain of possible solutions is

$$\prod_{v_i \in V} d(v_i)$$

where V is the set of variables in the goal literal (i.e., the literal obtained by instantiating the literal associated with the normal process with the substitution on the virtual input channel). Therefore, the number of possible solutions for N_1 messages is

$$N_1 \times \prod_{v_i \in V} d(v_i)$$

However, the actual number of solutions obtained is N_3. Therefore, the filter probability (FP) associated with this set of solutions is given by the equation

$$FP = \frac{N_3}{N_1 \times \prod_{v_i \in V} d(v_i)} \tag{21}$$

The filter tuple (FT) associated with this is $< n, FP >$, where n is the position of the literal associated with the process in the antecedents of the rule that generated the conjunct graph. Therefore,

$$FS_3 = FS_1 \cup \{FT\}$$

As an example, look in figure 30 at process "n(X1,Y)". In this case,

$$< N_1, S_1, NI_1, FS_1 >=< 2, \{X1 = uc2\}, 1, \{< 1, 1 >\} >$$

$$< N_2, S_2, NI_2, FS_2 >=< 2, \{Y = uc4\}, 2, \{\} >$$

Therefore, from equation 19, we have

$$S_o = Composition(\{X1 = uc2\}, \{Y = uc4\}) = \{X1 = uc2, Y = uc4\}$$

From equation 18, we have

$$NI_3 = NI_1 = 1$$

Also, equation 20 gives us

$$N_3 = N_2 = 2$$

The filter probability of the conjunct "n(X,Y)" for this compile-time message is given by equation 21

$$FP = \frac{N_3}{N_1 \times \prod_{v_i \in V} d(v_i)} = \frac{2}{2 \times 2} = 0.5$$

because the only variable in the goal is "Y" and its domain is 2. Therefore,

$$FS_o = \{< 1, 1 >, < 2, 0.5 >\}$$

3.3.6 Strategy for Dealing with Duplicate Solutions

Some rules can generate duplicate solutions to a goal. This can happen if a variable occurs in the tail of the rule but not in its head. For example, consider the rule

$$h(X,Z) : -t1(X,Y), t2(Y,Z)$$

If the subgoal $t1(X,Y)$ produced the two solutions $\{X = 3, Y = 5\}$ and $\{X = 3, Y = 6\}$ and the subgoal $t2(Y,Z)$ produced the two solutions $\{Y = 5, Z = 8\}$ and $\{Y = 6, Z = 8\}$, then $\{X = 3, Z = 8\}$ would appear twice as a solution for $h(X,Z)$. The communication estimation algorithm presented so far has to be modified if duplicates of this form are to be considered.

One more piece of information—*a duplication bag*—needs to be associated with each compile-time message. The *duplication bag* associated with a compile-time message includes the *duplication factors* of all the conjuncts that have been processed so far in the conjunct graph. A *duplication factor* for any particular conjunct in a conjunctive goal is a number, greater than or equal to one, and is a probabilistic measure of how many actual solutions are produced for each unique, actual solution of that conjunct. Therefore, if the duplication factor is 3 and we expect the total number of solutions generated to be 5, then we expect $5/3 = 1.67$ of them (probabilistically) to be unique. In the example given above, the literal goal that is solved by the given rule would have the duplication factor 2.0 associated with the solution $\{X = 3, Z = 8\}$. Conjuncts that are ancestors along more than one path will have as many copies of their *duplication factors* in the *duplication bag*. This is the reason a *duplication bag* is a *bag* and not a *set*. Just like a *filter set*, a *duplication bag* contains 2-tuples, one for each literal in the antecedent of the rule that generated the associated conjunct graph. Each tuple contains: (1) a number indicating the position (leftmost being 1) of the literal in the antecedents of the rule and (2) the duplication factor for the literal that led to the set of substitutions associated with the compile-time message.

Just as equation 11 (reproduced below as equation 22) led to a particularly simple formulation of the Communication Estimation algorithm for the no duplicate solutions case, there is a similar equation for the duplicate solutions case.

$$N = \prod_{v_i \in V} d(v_i) \times \prod_{i=1}^{n} FP(C_i) \tag{22}$$

The number of solutions N for a set of conjuncts (taking duplicates into account) is given by the equation below:

$$N = \prod_{v_i \in V} d(v_i) \times \prod_{i=1}^{n} FP(C_i) \times \prod_{DF_i \in DFB} DF_i \tag{23}$$

where V is the set of variable in the conjunctive goal, $d(X)$ gives the size of the domain of variable X, $FP(C_i)$ is the filter probability of conjunct C_i, and DFB is the *duplication factor bag* of the set of conjuncts. The *duplication factor bag* is the bag of duplication factors associated with the set of conjuncts. The number of instances of each duplication factor in the *duplication factor bag* is the number of distinct paths from the associated conjunct to the Tail process associated with the conjunctive goal. For example, in figure 31, there are 2 instances of the duplication factor associated with "a" and 1 instance each of the duplication factors associated with "b" and "c" in the *duplication factor bag* for the conjunctive goal. If the *duplication factors* of the 3 conjuncts a, b, and c are DF_1, DF_2, and DF_3 respectively, then the *duplication factor bag* for the conjunctive goal is $\llbracket DF_1, DF_1, DF_2, DF_3 \rrbracket$ In this case,

$$\prod_{DF_i \in DFB} DF_i = DF_1^2 \times DF_1 \times DF_3$$

In particular, if $DF_1 = 2$, $DF_2 = 1$, and $DF_3 = 1$, then

$$\prod_{DF_i \in DFB} DF_i = 2^2 \times 1 \times 1 = 4$$

In other words, four copies should be expected (on the average) for each unique solution of the conjunctive goal.

Again, just as in the Communication Estimation Algorithm (with no duplicates), the algorithm with duplicates follows naturally from the formulation of the problem given in this section. The detail in section 3.3.7 may be skipped safely on the first reading of the dissertation with no loss of continuity. Interested readers may return later for more detail.

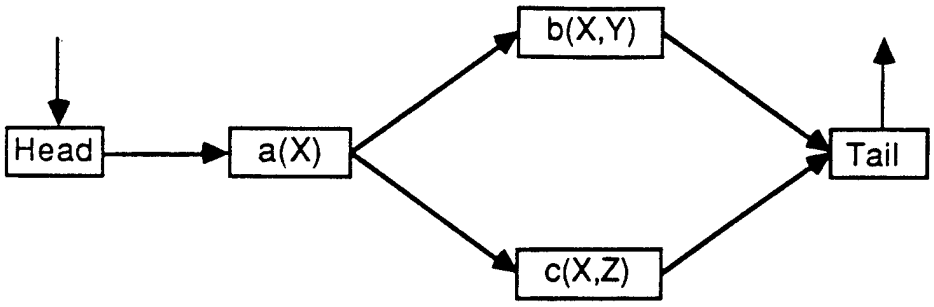


Figure 31: A Conjunctive Goal

3.3.7 Communication Estimation Algorithm (with Duplicates)

The description of the algorithm is divided into four parts just as it was done for the no duplicate case. Also, only the differences from the no duplicate case will be explained.

3.3.7.1 Analog of the *CP* function

There are n input channels and the messages on the channels are

$$< N_i, S_i, NI, FS_i, DB_i >$$

In case, the messages contain inconsistent substitutions, then the output is $\perp$ as before. If the output is not $\perp$, it is a message

$$< N_o, S_o, NI, FS_o, DB_o >$$

Notice that the initial number of substitutions is the same for the input messages as the output message because all of them belong to the same conjunct graph.

As before,

$$FS_o = \bigcup_{i=1}^{n} FS_i$$

However,

$$DB_o = \bigoplus_{i=1}^{n} DB_i$$

where the symbol $\oplus$ denotes *bag sum.*

The expected number of solutions for each initial substitution of the conjunct graph, N_{int}, is given by an application of equation 23:

$$N_{int} = \prod_{v_i \in V} d(v_i) \times \prod_{FP_i \in FPS} FP_i \times \prod_{DF_i \in DFB_o} DF_i$$

where DFB_o, the *duplication factor bag*, is the bag of duplication factors in the duplication tuples belonging to DB_o. For example, if the duplication bag is $[\![< 1,1.5 >, < 1,1.5 >, < 2,3.5 >]\!]$ where 1.5 is the duplication factor for the first conjunct and 3.5 is the duplication factor for the second conjunct, then the duplication factor bag is $[\![1.5, 1.5, 3.5]\!]$ Notice the slight variation from equation 13 for the no duplicate case.

The formula above gave the expected number of solutions for each initial substitution. Therefore, the total expected number of solutions, N_o, is given by

$$N_o = NI \times \prod_{v_i \in V} d(v_i) \times \prod_{FP_i \in FPS} FP_i \times \prod_{DF_i \in DFB} DF_i$$

3.3.7.2 Response of Normal Process to Compile-Time Messages on Virtual Input Channel

Let the compile-time message on the virtual input channel be

$$< N_i, S_i, NI_i, FS_i, DB_i >$$

and the compile-time message on the subtask channel be

$$< N_o, S_o, NI_o, FS_o, DB)_o >$$

In case, a rule is used to reduce the goal,

$$N_o = N_i \times PU$$

where PU is the probability of unification of the goal with the head of the rule.

In case, a fact, with an associated number of NF, is used to reduce the goal,

$$N_o = N_i \times PU \times NF$$

where PU is the probability of unification of the goal with the fact. Again, we have

$$NI_o = N_o$$

Also,

$$S_o = \{\}$$

$$FS_o = \{\}$$

$$DB_o = [\![\,]\!]$$

$[\![\ ]\!]$ stands for an empty bag.

3.3.7.3 Response of Tail Process to Compile-Time Messages on Virtual Input Channel

Let the compile-time message on the input channel be

$$< N_i, S_i, NI_i, FS_i, DB_i >$$

and the compile-time message on the task channel be

$$< N_o, S_o, NI_o, FS_o, DB_o >$$

As before,

$$N_o = N_i$$

$$S_o = Composition(IS, S_i)$$

$$NI_o = NI_i$$

$$FS_o = FS_i$$

In addition,

$$DB_o = DB_i$$

3.3.7.4 Response of Normal Process to Compile-Time Message on Subsolution Channel

Let the message on the virtual input channel to the normal process (that led to the creation of the subsolution in question) be

$$< N_1, S_1, NI_1, FS_1, DB_1 >$$

Also, let the compile-time message on the subsolution channel be

$$< N_2, S_2, NI_2, FS_2, DB_2 >$$

and the compile-time message on the output channel from the normal process be

$$< N_3, S_3, NI_3, FS_3, DB_3 >$$

As before,

$$N_3 = N_2$$

$$S_3 = Composition(S_1, S_2)$$

$$NI_3 = NI_1$$

The computation of the filter probability and duplication factor for the conjunct associated with the normal process is somewhat involved and needs additional notation. To make the notation easier to understand, a running example is used.

To begin with, let FP be the filter probability of the conjunct and DF be its duplication factor. Let the literal associated with the normal process be G and the rule used to reduce the goal be

$$G' : -SG'$$

where SG' is a set of conjuncts.

As our running example, consider the case where the rule is as given below.

$$h(a, X', Z', Q') : -t1(X', Y'), t2(Y', Z')$$

Therefore,

$$G' = h(a, X', Z', Q')$$

and

$$SG' = \{t1(X', Y'), t2(Y', Z')\}$$

Also, let G, the literal associated with the normal process, be as given below.

$$G = h(P, X, Z, W)$$

Assume that the domains of all the variables have cardinality 2.

Let *lvars* be the function that returns the set of variables in a literal. For example, $lvars(h(P, X, Z, W)) = \{P, X, Z, W\}$. Also, let *slvars* be the function that returns the set of variables in a set of literals. For example,

$$slvars(\{t1(X', Y'), t2(Y', Z')\}) = \{X', Y', Z'\}$$

Notation related to a goal: The goal to be solved is $G\ |_{S_1}$. VS_1, the set of variables in the goal, is given by the equation below.

$$VS_1 = lvars(G\ |_{S_1}) \tag{24}$$

For the running example, let $S_1 = \{W = b\}$. Therefore,

$$VS_1 = lvars(G\ |_{S_1}) = lvars(h(P, X, Z, b)) = \{P, X, Z\}$$

SD_1, the cardinality of the domain of solutions of the goal, is given by the equation below.

$$SD_1 = \prod_{v_i \in VS_1} d(v_i) \tag{25}$$

For the example,

$$SD_1 = d(P) \times d(X) \times d(Z) = 2 \times 2 \times 2 = 8$$

Notation related to an instance of a goal: An instance of the goal $G\ |_{S_1}$ that unifies with the head of the rule is solved. Let *mgu* represent the function that computes the most general unifier. Therefore, the most general unifier US of the goal $G\ |_{S_1}$ with the head of the rule G' is given by the equation below.

$$US = mgu(G_1, G') \tag{26}$$

For the example,

$$US = \{P = a, X' = X, Z' = Z, Q' = b\}$$

The instance of the goal that needs to be solved, is therefore

$$G\,|_{S_1}|_{US}$$

For the example, this goal instance is

$$h(a, X, Z, b)$$

VS_2, the set of variables in this instance of the goal, is given by the equation below.

$$VS_2 = lvars(G\,|_{S_1}|_{US}) \tag{27}$$

For the example,

$$VS_2 = \{X, Z\}$$

g, the cardinality of the domain of solutions of the instance of the goal, is given by the equation below.

$$g = \prod_{v_i \in VS_2} d(v_i) \tag{28}$$

For the example,

$$g = d(X) \times d(Z) = 2 \times 2 = 4$$

As before, the invocation substitution IS is the subset of the most general unifier US that contains bindings of variables in the goal $G\,|_{S_1}$ only and not the bindings of variables in G', the head of the rule.

For the example,

$$IS = \{P = a\}$$

Notation related to a conjunctive subgoal: As mentioned before, the rule in question is

$$G' : -SG'$$

SG_2, the conjunctive subgoal that needs to be solved, is given by the equation below.

$$SG_2 = SG' \mid_{US} \tag{29}$$

where US is the most general unifier of the goal and the head of the rule (as given in equation 26).

For the example,

$$SG_2 = \{t1(X, Y'), t2(Y', Z)\}$$

VS_4, the set of extra variables that are contained in SG_2, the conjunctive subgoal, and not in VS_2, the set of variables in the instantiated goal (see equation 27), is given by the equation below.

$$VS_4 = slvars(SG' \mid_{US}) - VS_2 = slvars(SG' \mid_{US}) - lvars(G \mid_{s_1} \mid_{US}) \tag{30}$$

For the example,

$$VS_4 = \{X, Y', Z\} - \{X, Z\} = \{Y'\}$$

h, the cardinality of the domain of these extra variables is given by the equation below.

$$h = \prod_{v_i \in VS_4} d(v_i) \tag{31}$$

For the example,

$$h = d(Y') = 2$$

Computation of filter probability and duplication factor: Let DF_2 be the compounded duplication factor of the conjuncts of the subgoal. Therefore,

$$DF_2 = \prod_{D_j \in DFB_2} D_j \tag{32}$$

where DFB_2 is the duplication factor bag associated with the duplication bag DB_2 (i.e., the set of duplication factors in the duplication bag DB_2). Remember that N_2 is the total number of solutions being reported. N_2/NI_2 gives the number of solutions for each initial substitution because NI_2 is the initial number of substitutions. Dividing N_2/NI_2 by DF_2 gives m, the number of unique solutions of the subgoal for each initial substitution. In other words,

$$m = \frac{N_2}{NI_2 \times DF_2} \tag{33}$$

Now, m unique solutions in the subgoal solution domain (cardinality $= g \times h$) are to be mapped into the instantiated goal domain (cardinality $= g$). For the example, the solutions for the subgoal are distributed over the cross-product of the domains of the variables in the set $\{X, Y', Z\}$. These are mapped into the cross-product of the domains of variables in the set $\{X, Z\}$. The problem is to find how many unique solutions will be obtained in the target domain. Since the distributions are random, the probability p' of a particular member of the instantiated goal domain not being one of the solutions is given by

$$p' = \frac{\begin{pmatrix} g \times h - h \\ m \end{pmatrix}}{\begin{pmatrix} g \times h \\ m \end{pmatrix}}$$

Therefore, the probability p that a particular member is one of the solutions is given by

$$p = 1 - p' = 1 - \frac{\binom{g \times h - h}{m}}{\binom{g \times h}{m}} \tag{34}$$

As pointed out by Treitel [69], the analysis given above is correct, strictly speaking, only when m is an integer. Since the value of m is an expected value based on a probabilistic analysis, it may not be an integer. Stirling's approximation for binomial coefficients can be used to solve this problem.

There is another problem that arises because the analysis above assumes that the value of m is known exactly as opposed to being an expected value. Since p is not a linear function of m, the expected value of p cannot be obtained simply by using the expected value of m in equation 34. This problem is ignored in this thesis.

Since the probability of a particular member of the instantiated goal domain being a solution is p and the size of the domain is g, the expected number of unique solutions in the domain is $p \times g$. Moreover, since the total number of solutions is m, the additional duplication factor due to this mapping (DF_a) is given by

$$DF_a = \frac{m}{p \times g}$$

The duplication factor for this solution to the goal, DF, is given by multiplying DF_2, the duplication factor for the subgoal solution (as given in equation 32), and this additional factor.

$$DF = DF_2 \times DF_a$$

The filter probability for this solution to the conjunct is obtained by dividing the number of unique solutions obtained ($= \frac{N_2}{DF}$) for each goal ($= \frac{N_2}{DF} \div N_1$) by the cardinality of the domain of possible solutions for the goal ($= SD_1$). Therefore,

$$FP = \frac{N_2}{N_1 \times SD_1 \times DF}$$

Algorithm	Complexity
Communication Estimation	Up to exponential factor less than run-time computation
Communication Cost Computation	$O(p^2)$
Communication Cost Recomputation	O(p)

p = Number of partitions

Table 2: Complexity Results for Communication Cost Computation

Now, DF and FP can be worked into the output message in the 2-tuple format. Let the position of the conjunct associated with the normal process be k. Therefore,

$$FS_3 = FS_2 \cup \{< k, FP >\}$$

$$DB_3 = DB_2 \oplus [\![< k, DF >]\!]$$

3.3.8 Complexity

Complexity results are summarized in table 2. More explanation including the basis for the results is given in the following sections (3.3.8.1 and 3.3.8.2).

3.3.8.1 Communication Estimation Algorithm

Using unknown constants in the abstract backward-chaining deduction ensures that the number of logical inferences in the abstract deduction is either equal to or less

than the number of inferences when no unknown constants are used. In the worst case, no reduction takes place in the number of logical inferences. In the best case, the number of logical inferences can be reduced by an exponential factor (as was seen earlier in section 3.2.

3.3.8.2 Communication Cost Computation Algorithm

If there are p partitions, complexity for this computation is $O(p^2)$ because all pairs of partitions may communicate with each other in the worst case. In case there are multiple copies, additional communication needs to be accounted for as described in section 3.1.4. However, this only takes a constant number of operations for each pair of partitions and therefore the complexity remains $O(p^2)$.

If the communication cost needs to be recomputed after a single partition is reallocated to another processor, the cost of the recomputation is $O(p)$ because the partition in question may communicate with all other partitions in the worst case. Again, the presence of multiple copies makes no difference to the complexity.

3.4 Processor Multiplexing Cost Computation

The computation of *processor multiplexing cost* (defined in section 3.1.5) is done by two algorithms. The first algorithm is called the *Processing Interval Assignment* algorithm. This algorithm performs an abstract simulation of *PM*, similar to the one for estimating communication. A side-effect of the simulation is the assignment of *processing intervals* for the operations that need to be performed. A *processing interval* is a 3-tuple of a *start time*, a *finish time* and a *processor load.* The second algorithm is called the *Processor Multiplexing Cost Computation* algorithm. This algorithm takes the output of the *Processing Interval Assignment* algorithm and an allocation and computes the *processor multiplexing cost.*

3.4.1 Cost Model

To estimate any cost, one needs a cost model. A cost model specifies the cost incurred for some set of basic operations. A useful cost model is one that picks these basic operations such that all operations that have any associated cost must be decomposable into these basic operations. This subsection presents a useful cost model for *PM*.

The basic operations chosen with their associated cost are:

- Selecting the next task to work on: We assume that any task that is ready to be executed may be picked. Cost assigned is 0.
- Selecting rules/assertions to unify with goal: This is essentially a database indexing operation. Cost is assumed to be a constant K_I.
- Plugging a substitution into a literal: Cost is K_P.
- Doing a successful unification: Cost is K_U.
- Doing an unsuccessful unification: Cost is K_{FU}.
- Doing a successful application of the *Merge* function: Cost assigned is 0.
- Doing an unsuccessful application of the *Merge* function: Cost assigned is 0.

Note that the constants used above are dependent on the multiprocessor used. These constants have units of time such as seconds, for example.

3.4.2 Processing Interval Assignment Algorithm

This algorithm is split into four parts just as the Communication Estimation algorithm was. In that algorithm, the response of processes to messages on different channels was described in terms of their communication requirements. We now do the same in the present algorithm in terms of processing requirements.

A compile-time message is augmented to include two other pieces of information: (1) A start time, ST, and (2) a finish time, FT. In its entirety, a compile-time message looks like:

$$< N, S, FS, DB, ST, FT >$$

Fields other than ST and FT have been defined before in section 3.3. It is assumed that the N actual messages that this represents are distributed uniformly in time from ST to FT. For the top-level goal, ST and FT are both 0. This is interpreted to mean that the top-level goal is given at time 0.

The basis for the probabilistic analysis here is the same as that for the Communication Estimation algorithm. All the detail that follows now for the four parts of the Processing Interval Assignment Algorithm can be safely skipped on the first reading without loss of continuity. Interested readers can return later for the additional detail.

3.4.2.1 Response of Normal Process to Compile-time Messages on Virtual Input Channel

Let the compile-time message on the virtual input channel be

$$< N_{in}, S_{in}, NI_{in}, FS_{in}, DB_{in}, ST_{in}, FT_{in} >$$

There are two cases that need to be considered. In the first case, there are n rules that may be used to reduce a goal. In the second case, NF facts may be used to solve the goal. These cases are treated separately. If there are both rules and facts to reduce/solve the goal, then it is easy to see how a combination of the two procedures may be used.

Case I: Rules only If there are n rules that may be applied to the goal, then n *subtask channels* will be set up and one message will be sent on each. We will assume that unifications with the rules are done in order from 1 to n. Assume also that the probability of unification of the goal with the k'th rule is PU_k. Let the compile-time message on the k'th subtask channel be

$$< N_{out_k}, S_{out_k}, NI_{out_k}, FS_{out_k}, DB_{out_k}, ST_{out_k}, FT_{out_k} >$$

Remember that the amounts of time taken for plugging in a substitution into a literal, for indexing the rules/assertions to unify with a goal, for a successful unification, and for an unsuccessful unification, are K_P, K_I, K_U and K_{FU} respectively. For each actual message to a *normal process*, the substitution in the message is applied to the literal associated with the process, all relevant rules/assertions are indexed, and unifications are attempted between the goal and the rules/assertions. Assume that there are n rules and unifications are attempted in order starting with the rule numbered 1 and ending with the rule numbered n. Therefore, $\triangle_k$, the time taken from the input of an actual message at a process to the possible output of a message on the k'th *subtask channel* (corresponding to the k'th rule) is given by:

$$\triangle_k = K_P + K_I + \sum_{i=1}^{k}[PU_i \times K_U + (1 - PU_i) \times K_{FU}] \tag{35}$$

Therefore, ST_{out_k} and FT_{out_k} are given by:

$$ST_{out_k} = ST_{in} + \triangle_k \tag{36}$$

$$FT_{out_k} = FT_{in} + \triangle_k \tag{37}$$

We will now characterize the *processing interval*, $< ST_I, FT_I, PL_I >$, associated with the processor for this computation. The *start time*, ST_I, of the processing interval is given by:

$$ST_I = ST_{in} \tag{38}$$

The *finish time*, FT_I, of the processing interval is given by:

$$FT_I = FT_{out_n} \tag{39}$$

Let PT be the the total amount of processing in time units for this computation.

$$PT = N_{in} \times \triangle_n \tag{40}$$

Therefore, the *processor load* PL_I, or the average number of virtual processors busy in the *processing interval*, is given by:

$$PL_I = \frac{PT}{FT_I - ST_I} \tag{41}$$

Plugging in the value of PT from equation 40 into equation 41, we get:

$$PL_I = \frac{N_{in} \times \Delta_n}{FT_I - ST_I} \tag{42}$$

Of course, all other fields of the output compile-time messages can be computed using the Communication Estimation algorithm.

Case II: Facts only In this case, NF facts are available for attempting to solve the goal. The compile-time message on the virtual input channel is

$$< N_{in}, S_{in}, NI_{in}, FS_{in}, DB_{in}, ST_{in}, FT_{in} >$$

as before. In this case, only one subtask channel is set up since the whole set of NF facts is considered in one pass (because they are included in one fact pattern). Let the compile-time message on the subtask-channel be

$$< N_{out}, S_{out}, NI_{out}, FS_{out}, DB_{out}, ST_{out}, FT_{out} >$$

As before, Δ_k, the time taken from the input of an actual message at a process to the (possible) output of a message on the *subtask channel* is given by:

$$\Delta_k = K_P + K_I + \sum_{i=1}^{k} [PU \times K_U + (1 - PU) \times K_{FU}] \tag{43}$$

Since only one subtask channel is set up, ST_{out} in this case is the minimum of the ST_{out_i}'s in the previous case (with rules only) and FT_{out} in this case is the maximum of the FT_{out_i} in the previous case. Therefore,

$$ST_{out} = ST_{in} + \Delta_1 \tag{44}$$

$$FT_{out} = FT_{in} + \Delta_{NF} \tag{45}$$

We will now characterize the *processing interval*, $< ST_I, FT_I, PL_I >$, associated with the processor for this computation. The *start time*, ST_I, of the processing interval is given by:

$$ST_I = ST_{in} \tag{46}$$

The *finish time*, FT_I, of the processing interval is given by:

$$FT_I = FT_{out} \tag{47}$$

Therefore, the *processor load* PL_I, or the average number of virtual processors busy in the *processing interval*, is given by:

$$PL_I = \frac{N_{in} \times \triangle_{NF}}{FT_I - ST_I} \tag{48}$$

3.4.2.2 Response of Tail Process to Compile-time Messages on Virtual Input Channel

Since no *basic operations* are included in this computation, no cost is incurred. A message on the virtual input channel produces a message on the *solution channel* immediately with no time delay.

3.4.2.3 Response of Normal Process to Compile-time Messages on Sub-solution Channels

Again, no *basic operations* are included and, therefore, the computation is free.

3.4.2.4 Analog of the *CP* function

Sim-Merge, the function described in the Communication Estimation algorithm, must be augmented further. The additional computation to be performed by *Sim-Merge* is described below.

Let there be n input channels and the messages on the channels be

$$< N_i, S_i, NI, FS_i, DB_i, ST_i, FT_i >$$

In case, the messages contain inconsistent substitutions, then the output is $\perp$ as before. If the output is not $\perp$, it is a message

$$< N_o, S_o, NI, FS_o, DB_o, ST_o, FT_o >$$

In this computation, no *basic operations* are included. However, one still has to assign a *start time*, ST_o, and a *finish time*, FT_o, to the output message. ST_o is the

earliest possible time that an actual message associated with the output compile-time message is sent out. Notice that there must be at least one actual message on each of the input channels to produce an actual message on the virtual input channel. Therefore,

$$ST_o = \max_{i=i}^{n} ST_i \tag{49}$$

Similarly, FT_o is the latest possible time that an actual message associated with the output compile-time message is sent out. Therefore,

$$FT_o = \max_{i=1}^{n} FT_i \tag{50}$$

A uniformity assumption has been made here that all actual messages associated with the output compile-time message are uniformly distributed over this interval from ST_o to FT_o.

3.4.3 Processor Multiplexing Cost Computation Algorithm

The algorithm will be referred to by its abbreviated name PMCCA. The input is a set of sets of *processing intervals*—one set for each processor that will be used at run-time. The output is a number that represents the *processor multiplexing cost* for the multiprocessor.

For this algorithm, processing intervals are represented in a different manner than before. Each processing interval is represented as two elements, one for each end-point of the interval. Some additional information is also included in each element. In all, an element is a 5-tuple with the following fields:

1. Type: This is either *start* or *finish* depending on whether this element represents the start end-point or finish end-point for the interval in question.

2. Time: This is the time associated with the start or finish end-point of the interval in question.

3. Load: This is the *processor load* associated with the *processing interval.*

4. CLoad: This is the cumulative load of all intervals that overlap at this instant in time.

This algorithm uses an auxiliary procedure *PMCCA-1* that takes the set of *processing intervals* associated with a single processor and returns the *processor multiplexing cost* for that processor only.[3] After this auxiliary procedure is run on each processor, the sum of all the individual *processor multiplexing costs* gives the *processor multiplexing cost* for the multiprocessor.

The procedure *PMCCA-1* uses an abstract data structure that we will call a *PQ-list*. The name suggests the similarity of the data structure to both priority-queues [3] and sorted lists. The elements are maintained in a 2-3 tree [3], for example, to get log performance for insertions and deletions. They are also maintained in a sorted list (in increasing order). This is easy because 2-3 trees have leaves in sorted order anyway from left to right. In all, the data structure supports the following abstract operations:

1. *InsertPQL(PQL, element, key)*: This inserts the element *element* into the PQ-list *PQL* in log time. In addition, the *CLoad* field of the element is set to the *CLoad* field of the previous element (in sorted order). If there is no previous element, then the field is set to zero.

2. *DeletePQL(PQL, element)*: This deletes *element* from *PQL* in log time.

3. *EnumeratePQL(PQL, element1, element2)*: Enumerates all elements in *PQL* in sorted order from the element *element1* to the element *element2*. This is done in time linear in the number of elements enumerated.

A detailed description of the procedure PMCCA-1 is given in appendix B. However, a rough description will be given here. Each processor has an associated *PQ-list* and a variable *PMC*. *PMC* is the current value of the *processor multiplexing cost* for the current set of processing intervals in the *PQ-list*. *PMC* is zero initially when there are no elements in the *PQ-list*. Each *processing interval* is inserted into the

[3]Notice that *processor multiplexing cost* is defined for a multiprocessor but a single processor is just a special case of a multiprocessor.

PQ-list as two elements using a procedure called *InsertPI* (for *Insert Processing Interval*). When elements are inserted into the *PQ-list*, the data-structure itself must be modified as necessary (by using the abstract operation *InsertPQL* for *PQ-lists*). In addition, the CLoad fields of all elements whose Time fields fall within the time interval of the processing interval may have to be modified. After all *processing intervals* have been inserted into the *PQ-list*, the value of *PMC* is the *processor multiplexing cost* for the processor.

Another procedure called *DeletePI* is used to remove processing intervals from the *PQ-list*. *DeletePI* is not used if the processor multiplexing cost is to be computed once only for a particular allocation. However, it is useful when more than one allocation needs to be considered. The next chapter will demonstrate this need.

Both *InsertPI* and *DeletePI* do a constant number of operations at most for every element in the *PQ-list* that lies between the two end-points (in sorted order). This can be verified by looking at the detailed description in appendix B. In the worst case, this set of elements could include every element in the *PQ-list*. In addition, the associated *InsertPQL* and *DeletePQL* operations take log time (in the number of elements in the *PQ-list*). Therefore, the total time complexity of both *InsertPI* and *DeletePI* is $O(n)$, where n is the number of elements in the *PQ-list*.

3.4.4 Complexity

The complexity results for the processor interval assignment algorithm and the processor multiplexing cost computation algorithm are summarized in table 3. More explanation including the basis for the results is given in the following sections (3.4.4.1 and 3.4.4.2).

3.4.4.1 Processor Interval Assignment Algorithm

Just as in the case of the communication estimation algorithm, an abstract backward-chaining deduction is done using unknown constants as the abstraction. Also, the number of additional operations for each logical inference is constant. Therefore, the complexity of this algorithm is the same as that for the communication estimation

Algorithm	Complexity
Processor Interval Assignment	Up to exponential factor less than run-time computation
Processor Multiplexing Cost Computation	$O(q\, r^2)$
Processor Multiplexing Cost Recomputation	$O(q\, r^2)$

q = Number of processors

r = Number of subgoals at compile-time

Table 3: Complexity Results for Processor Multiplexing Cost Computation

algorithm.

In fact, both the communication estimation algorithm and the processor interval assignment algorithm can be performed concurrently using just one abstract backward-chaining deduction. This is, in fact, how they are implemented. Although, this cost savings is important for an implementation, complexity results remain the same whether the two algorithms are performed concurrently or separately.

3.4.4.2 Processor Multiplexing Cost Computation Algorithm

Assume at first that no multiple copies are allowed for partitions. Let r be the number of subgoals generated during the abstract backward-chaining deduction of the *Processor Interval Assignment Algorithm.* Each subgoal will have an associated *processing interval.* In the worst case, all subgoals may be allocated to the same processor and, therefore, the same *PQ-list.* As mentioned before, *InsertPI* and *DeletePI* take $O(n)$ time, where n is the number of elements in the *PQ-list.* Therefore, the time taken for the combined set of *InsertPIs* to compute the processor multiplexing cost initially is $O(r^2)$.

Now, consider the case with multiple copies. Let q be the number of processors in the system. Now, the maximum number of copies possible for any partition is q. If the number of copies of a certain partition is m, then each of its processing intervals in the single copy case is now modelled as m processing intervals, each with $\frac{1}{m}$ of the original processor load. The time taken for this algorithm is the most when all r processing intervals have q copies associated, one in each processor. The combined set of *InsertPIs* will take $O(qr^2)$ time. Note that it is not $O(q^2r^2)$ because any single processor can only contain one copy of a partition.

Now, let us say that reallocations are allowed and a single partition may be reallocated to another processor. Processor multiplexing cost may be computed by using *DeletePIs* on the associated processing intervals to remove them from the original processor's *PQ-list* and then applying *InsertPIs* to insert the same processing intervals into the new processor's *PQ-list.* Since the partition in question

may include all the subgoals in the worst case, the cost of recomputation is $O(qr^2)$—the same as the worst case cost for the original computation. In the typical case, however, one would hope to do a lot better than this.

3.5 Summary

This chapter has presented the formal definition for the cost function that is the basis for allocation. The cost function relates well to intuitive notions of the quality of allocations. One way to view the cost function is that it treats all communication delays and delays due to sequentialization of parallel tasks as being on the critical path of the computation in the worst case. Since the parallel time for execution is the same for all allocations, it is the extra delay due to communication and sequentialization that should be used (and is used) as the cost function to compare different allocations.

An important feature of the cost function is that it is efficient to compute and recompute. Algorithms were presented to do this computation and recomputation.

The cost function ignores two aspects of allocations that should be included in a future allocator, if possible. First, as mentioned above, all delays and sequentializations of parallel tasks are considered to be on the critical path. It would be better to work without this assumption. Second, the communication delay function does not take congestion of communication channels into account. Despite these two simplifications, the cost function serves as a good basis for an allocator as the next chapter will show.

Chapter 4

Allocation Algorithms

This chapter describes the algorithms used by the allocator to perform a limited search of the space of allocations. In addition, the chapter includes experimental results obtained from an implementation of the allocator and *PM*.

There are two main algorithms for searching the space of allocations. Both use the cost function and associated algorithms described in the previous chapter. The first algorithm is a greedy algorithm in which partitions are allocated one at a time. A partition is allocated to the lowest cost processor without re-allocating any partitions that were allocated previously. The second algorithm is a local minimization algorithm. This algorithm consists of a sequence of cost-reducing re-allocations of partitions to neighboring processors.

Both allocation algorithms are described in detail next followed by experimental results. Some related work is also discussed at the end of the chapter.

4.1 Greedy Allocation

This section contains the specifications of the algorithm, a description of the algorithm, a discussion of its complexity, and an example to show that it does not necessarily produce a locally optimal solution. However, the section on experimental results will show later that, in a typical case, greedy allocation can produce good

allocations by itself.

4.1.1 Specifications

Inputs

1. P: a set of partitions of the database.
2. C: a function that takes two partitions P_1 and P_2 and returns a tuple of the form $< data, number >$ where *data* is the amount of data (in bytes) and *number* is the number of messages sent from partition P_1 to partition P_2. *data* and *number* are expected values in a probabilistic sense.
3. PI: a function that takes a partition and returns the set of processing intervals associated with the partition.
4. Multiprocessor constants: These are K_1, K_2, and K_3 used to compute communication cost as given by equations 5 and 6.
5. Topology: This includes (1) distances between all pairs of processors and (2) lists of neighbors of each processor.

Outputs

1. Allocation: A many-to-one mapping from the set of partitions to the set of processors.
2. Number of copies for each partition: If the number of copies is greater than 1, then the allocation above specifies the central processor for the cluster of copies. The number of copies will determine the processors around the central processor that will also contain copies of the partition.

4.1.2 Algorithm

Let us assume for now that each partition has a single copy. The extensions to handle multiple copies will be described later in this section.

The overall structure of the greedy allocation algorithm is as follows: Starting from an empty allocation, each partition is allocated one at a time. The single partition under consideration at any time is allocated to the processor that leads to the lowest cost. After a partition is allocated, it is not reallocated to another processor.

This algorithm is embodied in the procedure *GreedyAllocation* shown in figure 32. As shown in the figure, all inputs to the procedure are implicit. These inputs were described in the specifications to the algorithm given above. At the beginning of each iteration of the outer **For** loop, there is a partial allocation of some partitions to processors. Each iteration allocates the next partition to the processor that leads to the lowest cost. The inner **For** loop considers allocation of the partition to each processor in turn. The code segment "Allocate *Partition* to *Processor*" includes (1) the application of the procedure *InsertPI* from chapter 3 to each processing interval associated with the partition *Partition*—with the second argument of the call to *InsertPI* being the *PQ-list* associated with the processor *Processor*, (2) the update of the cost function due to the additional communication to/from the partition from/to those already allocated, and (3) the update of the state of allocation reflecting that the partition has been allocated to the processor. The code segment "Deallocate *Partition* from *Processor*" includes the opposite operations.

Multiple copies can be handled in a couple of ways. One method is more principled as well as more costly than the other one. I will describe this first. The only change required from the procedure *GreedyAllocation* is that the inner **For** loop needs to be changed as follows.

"**For** all *Processor* $\in$ *PotentialProcs* **do begin**"
needs to be changed to
"**For** all combinations of *Processor* $\in$ *PotentialProcs* and
NumCopies=1... *Cardinality(AllProcs)* **do begin**"
and
"Allocate *Partition* to *Processor*"
needs to be changed to

```
Procedure GreedyAllocation()
    begin
        PotentialProcs ← AllProcs;
        /* AllProcs is the set of all processors */
        For all Partition ∈ SetOfPartitions do begin
        /* SetOfPartitions is the set of all partitions */
            BestCost ← ∞;
            BestProc ← nil;
            For all Processor ∈ PotentialProcs do begin
                Allocate Partition to Processor;
                TempCost ← Cost(Allocation);
                If Tempcost < BestCost then begin
                    BestCost ← TempCost;
                    BestProc ← Processor
                end; /* If */
                Deallocate Partition from Processor
            end; /* For */
            Allocate Partition to BestProc
        end /* For */
    end; /* GreedyAllocation */
```

Figure 32: Procedure *GreedyAllocation*

"Allocate *NumCopies* copies of *Partition* to *Processor*."

This last statement is interpreted to mean that the central processor of the cluster of the *NumCopies* copies should be *Processor*. Call this modified version of the procedure *GreedyAllocation'*.

Another way to handle multiple copies is to decide the numbers of copies of all partitions prior to using the procedure *GreedyAllocation*. The numbers can be picked heuristically. One reasonable way to pick the number of copies of a partition is to take the highest degree of parallelism exhibited by the partition. The highest degree of parallelism is simply the maximum of the *processor-load* function associated with the partition as described in chapter 3. Since the number of copies cannot be any arbitrary number, and certainly not a fractional number, the number of copies is picked arbitrarily to be the next higher acceptable number greater than the maximum degree of parallelism. Call this modified version of the procedure *GreedyAllocation''*. This method is less expensive than the first one. Actual complexities of the two methods will be compared in the next section.

Notice that in the code for the procedure *GreedyAllocation* shown in figure 32, no mention was made of the order in which partitions are chosen for allocation out of the set *SetOfPartitions*. In practice, the order of allocation can affect the allocation chosen by the procedure. The order that is used in this thesis is the topological order associated with the dataflow* graph of the computation. If a partition occurs multiple times in a topological search, its first instance is chosen for the ordering. This order of allocation ensures that partitions are allocated only after previously used partitions in the dataflow* graph have been allocated, thereby giving the greedy allocation procedure some context in which to make reasonable decisions. Prior to using the topological ordering, a random ordering was used and discarded because it would make bad allocations for partitions that did not have any communicating partitions allocated before it.

In the special case when communication delays are assumed to be zero, there is an even more effective order of allocation. In particular, Graham [29] has shown that a particular order gives an upper bound on completion time of twice the optimal

Algorithm	Complexity
GreedyAllocation	$O(p^2q+pqr^2)$
GreedyAllocation'	$O(p^2q^2+pq^3r^2)$
GreedyAllocation''	$O(p^2q+pq^2r^2)$

p = Number of partitions

q = Number of processors

r = Number of subgoals at compile-time

Table 4: Complexity Results for Greedy Allocation

completion time (asymptotically when the number of processors goes to infinity). In this ordering, the next task chosen for execution at any time out of a DAG of tasks is always the one that "heads the *longest chain* of unexecuted tasks (in the sense that the sum of the task times in the chain is maximal)." Unfortunately, this result does not apply to the case where communication delays are non-zero.

4.1.3 Complexity

The complexity results for greedy allocation are summarized in table 4. Further explanation including the basis of the results is given below.

Let p be the number of partitions, q the number of processors, and r the number of subgoals in the dataflow* graph generated during abstract backward-chaining

deduction for the Processing Interval Assignment algorithm as well as the Communication Estimation algorithm.

The time to update the communication cost function when a single partition is allocated to a single processor is p (see section 3.3.8). The time to update processor multiplexing cost when a single partition is allocated is r^2 when only single copies are allowed and it is qr^2 when multiple copies are allowed (see section 3.4.4.2). The combined cost is $O(p + r^2)$ for single copies and $O(p + qr^2)$ for multiple copies. Deallocation leads to the same cost and, therefore, the order of complexity for a combined allocation and deallocation is the same as simply an allocation.

The outer loop is executed p times—once for each partition. For *GreedyAllocation* as well as *GreedyAllocation''*, the inner loop is executed q times. For *GreedyAllocation'*, the inner loop is executed q^2 times since the cardinality of *PotentialProcs* in the worst case (q) multiplied by the cardinality of *AllProcs* (q) is q^2. Therefore, *GreedyAllocation* and *GreedyAllocation''* require pq updates due to allocations and *GreedyAllocation'* requires up to pq^2 updates due to allocations. Multiplying the number of updates by the complexity of each update gives the complexity of the entire algorithm. Therefore, the complexity of *GreedyAllocation* is $O(pq \times (p + r^2))$, which is $O(p^2q + pqr^2)$. Similarly, the complexity of *GreedyAllocation''* is $O(pq \times (p + qr^2))$, which is $O(p^2q + pq^2r^2)$. Finally, the complexity of *GreedyAllocation'* is $O(pq^2 \times (p + qr^2))$, which is $O(p^2q^2 + pq^3r^2)$.

An optimization is possible for the greedy allocation procedures that can reduce the absolute cost of the procedures but does not affect the worst case complexity measures derived above. *PotentialProcs* in the procedures need not be *AllProcs*. In the single copy case, for example, allocations need to be considered only to processors that already have partitions allocated to them or their neighbors. As a special case, the first partition should be allocated immediately to the processor where the computation will begin (which is assumed to be the same as the processor where the final result will be demanded). When partitions can have multiple copies, this gets a bit more involved but the general idea is the same. Notice that this optimization does not reduce the size of *PotentialProcs* in the worst case, which is q.

4.1.4 Not Locally Optimal

Once allocated to a processor, a partition is not re-allocated to another processor when allocations of other partitions are being considered. This is done regardless of any new communication requirements that the later partitions may expose. Therefore, it is not surprising that greedy allocation is not guaranteed to produce a locally optimal allocation. An example of greedy allocation that does not produce a locally optimal solution is given in appendix C. Of course, if the solution is not locally optimal, it is also not globally optimal.

4.2 Local Minimization

This section contains a specification of the algorithm, a description of the algorithm, a discussion of its complexity, and an example to show that the allocations produced are not necessarily globally optimal.

4.2.1 Specifications

Inputs

1. P: a set of partitions of the database.

2. C: a function that takes two partitions P_1 and P_2 and returns a tuple of the form $< data, number >$ where *data* is the amount of data (in bytes) and *number* is the number of messages sent from partition P_1 to partition P_2. *data* and *number* are expected values in a probabilistic sense.

3. PI: a function that takes a partition and returns the set of processing intervals associated with the partition.

4. Multiprocessor constants: These are K_1, K_2, and K_3 used to compute communication cost as given by equations 5 and 6.

5. Topology: This includes (1) distances between all pairs of processors and (2) lists of neighbors of each processor.

6. An allocation: A many-to-one mapping from the set of partitions to the set of processors.

7. Number of copies for each partition

Outputs

1. Allocation: A many-to-one mapping from the set of partitions to the set of processors.

4.2.2 Algorithm

Notice from the specifications given above that the number of copies for each partition is already fixed by the greedy allocation procedure. In fact, the number of copies is an input to the procedure.

The code for the local minimization procedure *LocalMinimization* is given in figure 33. The idea is that there is a set of iterations specified by the outer **While** loop. In each iteration, every partition is considered in turn (by the outer one of the two nested **For** loops. The best allocation is picked for each partition among the processor it is currently currently allocated to and its neighbors—six in the case of FAIM-1. This is done in the inner **For** loop. At the conclusion of the inner **For** loop, the partition is allocated to the best processor among the ones considered. If this is different from the processor that the partition was allocated to, then the boolean variable *Changed?* is set to **true**. Therefore, *Changed?* gets set to **true** if one or more partitions get reallocated to a neighboring processor. The **While** loop terminates when *Changed?* is **false**, or equivalently when no partitions were reallocated in the previous iteration of the **While** loop.

```
Procedure LocalMinimization()
    begin
        Changed? ← true;
        While Changed? = true do begin
            Changed? ← nil;
            For all Partition ∈ SetOfPartitions do begin
                CurrProc ← Processor(Partition);
                BestCost ← Cost;
                BestProc ← CurrProc;
                Deallocate Partition from CurrProc;
                For all Processor ∈ Neighbor(CurrProc) do begin
                    Allocate Partition to Processor;
                    TempCost ← Cost(Allocation);
                    If TempCost < BestCost then begin
                        BestCost ← TempCost;
                        BestProc ← Processor
                    end; /* If */
                    Deallocate Partition from Processor
                end; /* For */
                If BestProc ≠ CurrProc then begin
                    Allocate Partition to BestProc;
                    Changed? ← true
                end /* If */
            end /* For */
        end /* While */
    end; /* LocalMinimization */
```

Figure 33: Procedure *LocalMinimization*

Algorithm	Complexity	
	Single Copies	Multiple Copies
LocalMinimization	$O(q^p(p^2+pr^2))$	$O(q^p(p^2+pqr^2))$
One iteration of While loop in *LocalMinimization*	$O(p^2+pr^2)$	$O(p^2+pqr^2)$

p = Number of partitions

q = Number of processors

r = Number of subgoals at compile-time

Table 5: Complexity Results for Local Minimization

4.2.3 Complexity

Table 5 summarizes the complexity results for Local Minimization. Further explanation including the basis for the results is given below.

In the worst case, *LocalMinimization* may consider all possible allocations. These are exponential in number. To be precise, there are q^p allocations, where p is the number of partitions and q is the number of processors. Notice that after each iteration of the **While** loop, there is always a complete allocation that is the lowest cost allocation found so far. As it turns out, each iteration takes polynomial time (see below). Therefore, if the algorithm has exceeded some time limit, it can be terminated between iterations of the **While** loop and the latest allocation can be used.

The time taken for each iteration of the **While** loop can be analyzed as follows.

Each partition is allocated (and deallocated) 7 times in each iteration of the While loop. The cost for updating the cost function for each allocation/deallocation is $O(p + r^2)$ when single copies of partitions are used and it is $O(p + qr^2)$ when multiple copies of partitions are allowed (see previous discussion on complexity of *GreedyAllocation*). Since there are p partitions, the total times taken for an iteration are $O(p^2 + pr^2)$ and $O(p^2 + pqr^2)$ for the single copy and multiple copy cases respectively.

4.2.4 Not Globally Optimal

Even if the procedure *LocalMinimization* is executed till it terminates (as opposed to just a few rounds), there is no guarantee that the locally optimal allocation is going to be globally optimal as well. Appendix D contains an example of an allocation produced by *LocalMinimization* that is not globally optimal.

4.3 Experimental Results

PM, the parallel execution model, and the resource allocation algorithms have been implemented in Zetalisp on the Symbolics 3600 series of Lisp Machines [44].[1] *PM* and the simulated version of *PM* were implemented on top of a high-level functional simulation of FAIM-1 using the event-driven simulator Helios [24]. The parallel interpreters were created by modifying the sequential backward-chaining interpreter in MRS [54], a logic programming system.

Several examples have been tried using this implementation. One of these will be described in detail to demonstrate the utility of *PM* and the resource allocation techniques developed in this thesis. The example logic program describes the structure and behavior of a digital device—a 4-bit adder. In addition, a set of facts describes the values of all the inputs. The goal given to the backward-chaining deduction engine is to determine the value of a particular output. This problem is similar, but not identical, to a part of the problem of test-generation [59]: the

[1] Zetalisp and Symbolics are trademarks of Symbolics, Inc.

determination of values for a set of inputs that would force an output (or some other intermediate port) to a particular value.

Detailed information about the example is given in appendix E. In particular, the appendix contains the complete database for the example, the goal given to the backward-chaining engine, the partitioning of the database, the FAIM-1 multiprocessor configuration used, other multiprocessor parameters, and finally the allocations generated by the allocator. Two allocations are shown: the first for the single copy case and the second for the multiple copy case.

Figure 34 shows the parallelism profile for the application. The profile gives the number of parallel inferences versus time assuming unbounded processors and memory, and instantaneous communication. The figure shows two curves: the curve marked "AOP" shows the profile when *and-parallelism*, *or-parallelism*, and *pipelining* are exploited and the curve marked "OP" shows the profile when only *or-parallelism* and *pipelining* are exploited. The average and maximum parallelism for the "AOP" case are 30.371 and 106 respectively. The same numbers for the "OP" case are 12.745 and 37 respectively. The numbers demonstrate the advantage of exploiting and-parallelism.

The same curves also give unreachable lower bounds on the time to complete the computation. The lower bound is simply the maximum time value for the curve. In any real multiprocessor, the completion time will be greater than this lower bound because it will have only a limited number of processors (as opposed to an unlimited number assumed here) and non-zero communication delays (as opposed to instantaneous communication assumed here). The lower bound for the "AOP" case is 35 logical inference time units and the lower bound for the "OP" case is 51 logical inference time units. Again, these numbers indicate the advantage of exploiting *and-parallelism*.

The curves also give the sequential time for computation. The sequential time is simply the area under the curve. The sequential time for the "AOP" case is 1063 logical inference time units and the sequential time for the "OP" case is 650 logical inference time units. Notice that the sequential time for the "OP" case is lower than the sequential time for the "AOP" case. Therefore, if only one processor

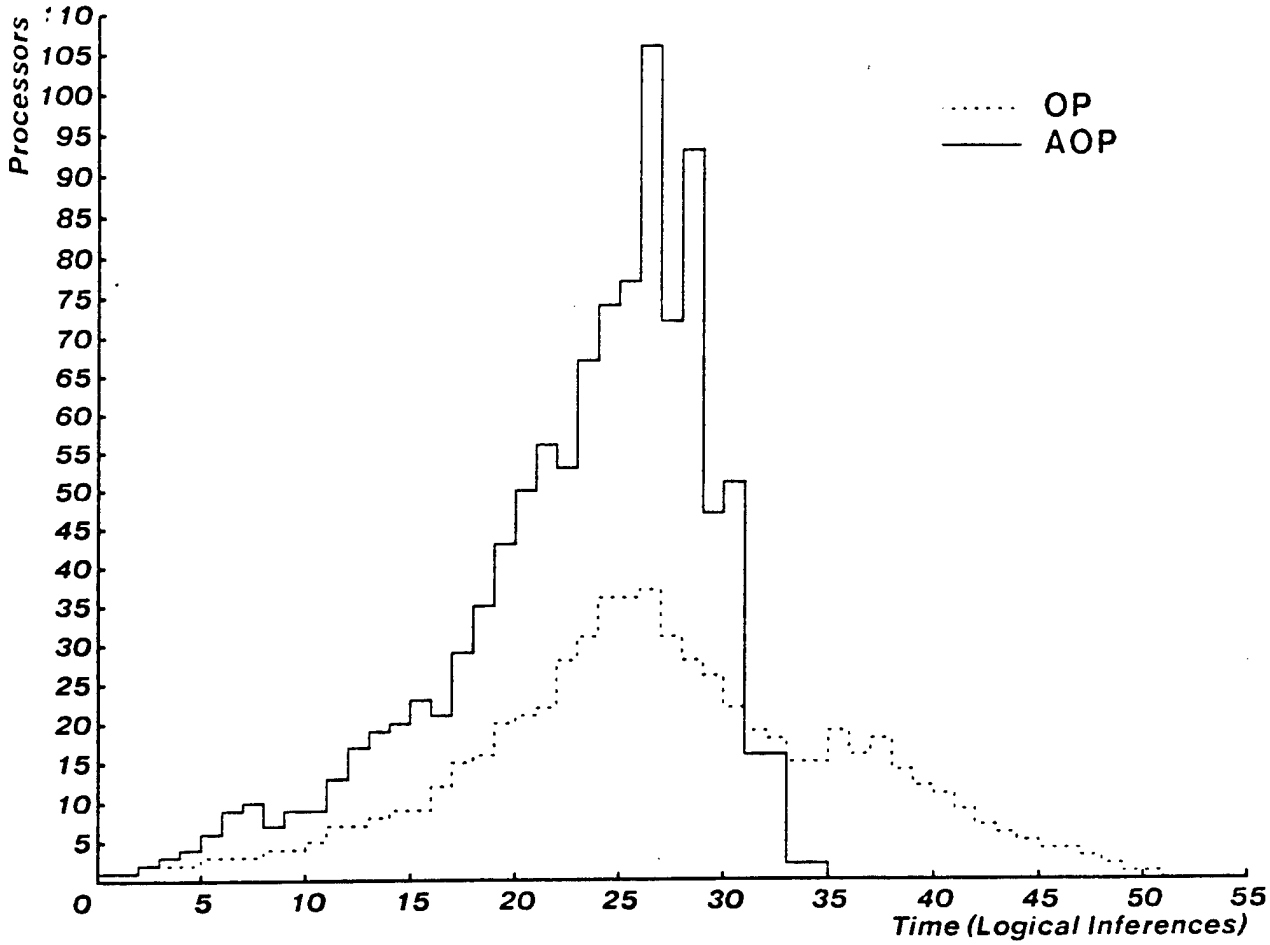


Figure 34: Parallelism Profile for Adder Example

is available, it is more efficient to exploit less parallelism conceptually. Of course, it could have been the other way around also as pointed out in chapter 2. The corresponding unreachable upper bounds on speedups for "AOP" and "OP" can be computed by dividing the sequential time by the unreachable lower bound on time taken in the parallel case. These upper bounds are 1063/35 (= 30.371) and 650/51 (= 12.745) "AOP" and "OP" respectively. Notice that these unreachable speedup numbers are the same as the average parallelism numbers given earlier (as they should be).

Earlier experiments with smaller examples had indicated that greedy allocation by itself either produced locally optimal allocations or allocations that were very close to locally optimal. The experiments described for the adder example use greedy allocation only; no local minimization was used.

A possible explanation for greedy allocation turning out to be so successful is given now. The only hand-designed situations where greedy allocation performs poorly are cases where the communication and processing requirements for a dataflow* graph are highly non-uniform (see appendix C for an example). In the practical examples looked at, this was not the case (i.e., processing and communication requirements were fairly uniform). In particular, for the adder example being considered here, all communication arcs in a conjunct graph carry a single message, if they carry one at all. This follows directly from the fact that the output of a hardware component is a function of the inputs. In addition, the amount of processing associated with nodes in a conjunct graph is fairly uniform. The number of rules that apply to reducing any particular goal ranges from two to four only.

When *GreedyAllocation* was used to make a single copy allocation, the time taken and speedup were found to be 215.531 logical inference time units and 4.932 respectively.

While using the single-copy allocation generated by the allocator program, it was noticed that certain partitions were bottlenecks in the computation. The first clue came from monitoring the "busy-ness" of various processors during the parallel computation.[2] The second clue came from looking at the parallelism profile of

[2]This was done by using a color instrumentation tool in Helios, the event-driven simulator. A

single partitions. Some partitions came out with very high parallelism for some time intervals indicating that they might be bottlenecks.

Given the evidence of a bottleneck due to single copies, it was decided to allow multiple copies in the allocation. The procedure *GreedyAllocation''* was used alone without any local minimization. If local minimization were used, it would only improve on this allocation. It turns out that the time taken and speedup for the allocation generated were 60.254 logical inference time units and 17.642 respectively. Compared to the single copy case, the speedup is a multiple of 3.577 higher.

A random allocator was used to generate an allocation using the same number of copies for each partition as that used by *GreedyAllocation''*. Time taken and speedup were 215.531 logical inference time units and 17.015 respectively.

GreedyAllocation'' is not much better than a random allocator in this case because communication is relatively cheap in the FAIM-1 multiprocessor configuration considered. However, there are at least two cases where a random allocation can perform arbitrarily worse than a greedy allocation. Both of these two cases have the characteristic that the average delays in the random allocation case are arbitrarily larger than the average delays expected in the greedy allocation case. The first case is one in which there are a larger number of processors. A larger number of processors increases the average distance between a random pair of processors. This increases the expected distance for communication using a random allocation. However, greedy allocation does not use more processors unless that decreases the cost function. In other words, adding more processors does not necessarily mean that they will be used by greedy allocation. The second case is one in which different communication hardware is used and communication is higher even for the same distances as before. This could happen if a different multiprocessor were used that did not have a high degree of hardware support for communication (as it is for FAIM-1).

Now, it remains to be seen what the effect of higher delays is on random allocations. Figure 35 illustrates this effect. The figure plots speedup versus log (base

color spectrum from blue to red was used to indicate the "busy-ness" of processors represented by icons, with red being used to indicate the busy extreme and blue being used to indicate the idle extreme.

2) of delay (expressed as a multiple of the normal delay expected for the FAIM-1 configuration) for a set of experiments performed using the random allocation mentioned above. Delays to the left of the speedup axis are sub-normal delays (down to 2^{-10} times the normal delay) and delays to the right are super-normal delays (up to 2^{10} times the normal delay). The relative flatness of the curve to the left of the speedup axis demonstrates that communication is not a bottleneck in this case. However, as communication delays are increased beyond the normal delays, the speedup for random allocation drops to zero asymptotically. Let us see how a greedy allocation might perform in the two cases in which delays are increased. When the number of processors is increased, the speedup expected from greedy allocation should be as good or better than 17.642 (the speedup for the configuration used for the greedy allocation experiment mentioned earlier). For the random allocation case, a delay that is 4 times normal drops speedup to about 12.5. Given the topology of FAIM-1 and the multiprocessor communication constants, it turns out that this delay would be expected when the number of processors is increased to about 4000. Let us look at the other case now. If communication delays are higher overall for the multiprocessor, then communication cost will overwhelm processor multiplexing cost beyond a certain point. Therefore, all computation will get allocated to a single processor and speedup will be 1. In the random allocation case, however, it could be arbitrarily close to zero. As a somewhat less extreme case, a delay of 128 times the normal FAIM-1 delay drops the speedup below 1 (see figure). This can easily happen if the multiprocessor does not have the type of specialized communication support that FAIM-1 has.

On a different note, it was mentioned in section 3.1.4 that a possible improvement in the communication cost might be to reduce it by the degree of communication parallelism. There is some evidence that this might be true. A reasonable measure of the degree of communication parallelism for a computation might be the average parallelism given by its parallelism profile (assuming that the degree of communication parallelism is the same as the degree of processing parallelism). In the case of the adder example, this is 30.371. An allocation was produced by reducing the

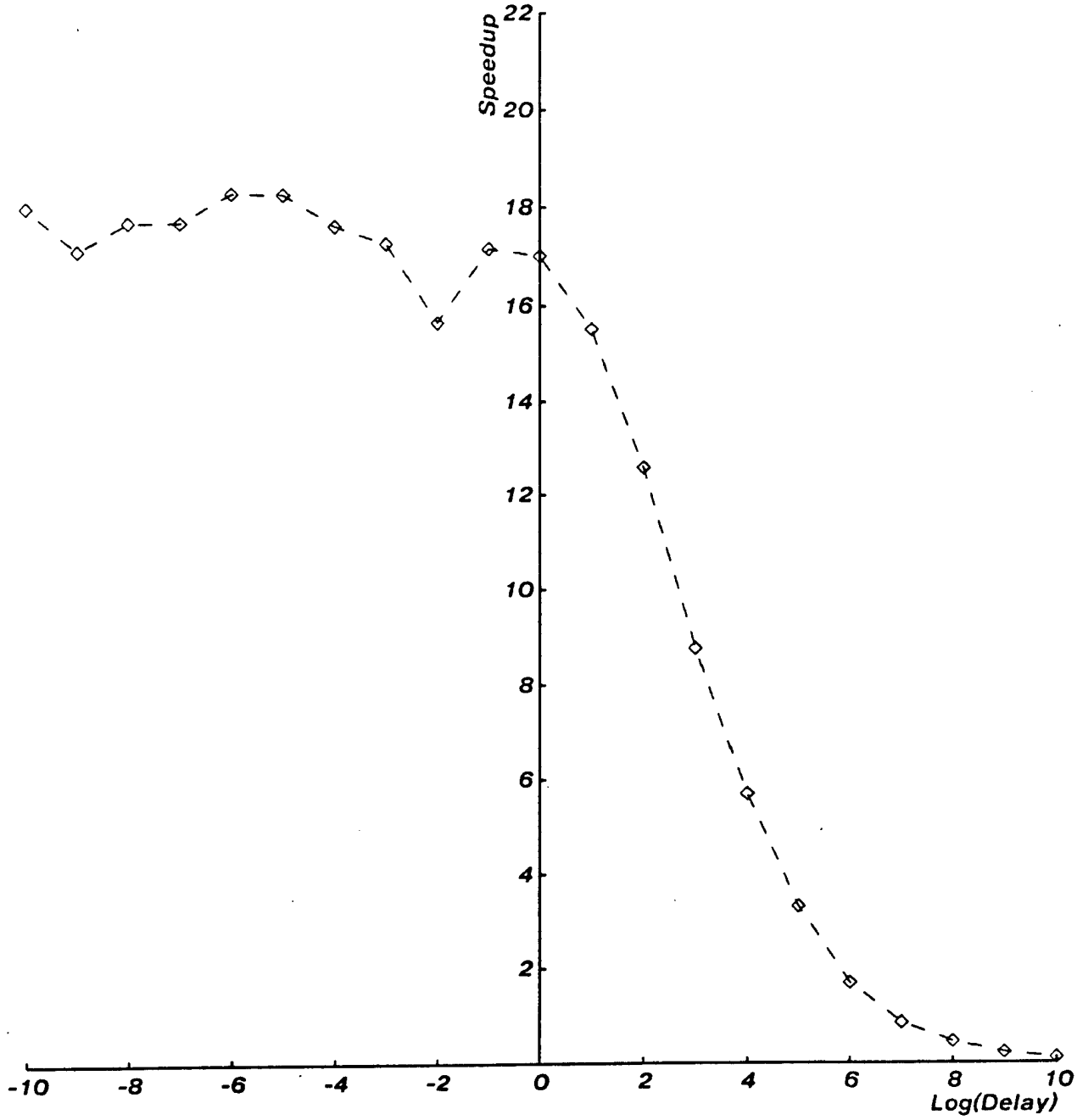


Figure 35: Speedup vs. Delay for Random Allocation

communication parameters (K_1, K_2, and K_3 in equation 1, section 3.1.4) by a factor of 32 (closest power of 2 to 30.371). This allocation produced by the procedure *GreedyAllocation''* gives an average speedup of 18.250 as opposed to 17.642 with the normal communication parameters. The speedup did improve by taking communication parallelism into consideration. However, this single data point should be considered as suggestive evidence only. Conclusive proof can only be provided by further research. Of course, there may be more accurate methods to take communication parallelism into account and the associated speedup improvement may be even greater.

4.4 Related Work

4.4.1 Theoretical work

Previous theoretical work on scheduling (or allocation) for multiprocessors [37,39] is not directly applicable here. There are many variations on the scheduling problem but none of them include communication cost in a general way. There are many interesting results, however, that may be good starting points for extensions that consider communication. Extensions to approximation results such as Graham's [29] would be tremendously useful. Another extension that would be required to attack the scheduling problem in this thesis would be the inclusion of memory constraints that limit the number of copies of certain pieces of the database (or code in procedural languages).

4.4.2 Local Search

The local minimization algorithm discussed in this chapter is an application of a general technique called *Local Search* in the optimization literature (see book by Papadimitriou and Steiglitz [51], for example). The general algorithm is described in the book by Papadimitriou and Steiglitz as follows:

Given an instance (F,c) of an optimization problem, where F is the feasible set

and c is the cost mapping, we choose a neighborhood

$$N : F \longrightarrow 2^F$$

which is searched at point $t \in F$ for improvements by the subroutine

$$\text{improve}(t) = \begin{cases} \text{any } s \in N(t) \text{ with } c(s) < c(t) \text{ if such an } s \text{ exists} \\ \text{"no" otherwise} \end{cases}$$

The book contains many examples of local search algorithms applied to the travelling salesman problem and the uniform graph partitioning problem among others. In addition, the book identifies some general issues in the development of such algorithms. In many cases, local search has turned out to be a powerful optimization technique and is often the best available. Unfortunately, the development of local search algorithms remains largely an art and the demonstrations of utility are empirical in nature.

Recall that in this thesis, the local minimization algorithm turned out not to be very important. The starting point for local minimization (i.e., the result of greedy allocation) was already quite good.

4.4.3 Compile-time Allocation for Dataflow

4.4.3.1 DDM2 from University of Utah

A paper by Martha Chamberlain and Alan Davis [11] describes what was probably the first attempt at static allocation of dataflow programs. The target machine was called DDM2 (a successor to DDM1) and a single processor version was operational in 1979. Timing measurements taken from the single processor version were then used to emulate a multiple processor version whose topology was a tree.

The input to the allocator is a type of dataflow graph called DDN (Data Driven Net). The overall goal of the allocator was to massage this graph into a tree-structured shape preserving as much of the locality as possible. Function-preserving graph transformations such as replicating nodes and inserting dummy nodes for extra synchronization were used.

The overall structure of the allocator consists of three top-level steps. First, the DDN is converted to a TANTA graph (or Two-terminal, Acyclic graph with No Transitive Arcs). Since DDN's are already two-terminal, this phase consists of encapsulating cyclic iteration structures into single complex nodes and removing transitive arcs. The second top level step is the conversion of TANTA graphs to SP graphs (or series-parallel graphs). Different methods to do this lead to minimum work or minimum time (i.e., minimum critical path). The third and final step is to convert the SP graph to a tree by a series of folding operations.

In comparison with the allocator presented in this thesis, a lot of processing in the DDM2 allocator is geared specifically towards the special-purpose tree topology. The allocator in this thesis is not designed for any particular topology. Another point of difference is that the DDM2 allocator makes the simple assumption of equal computation cost for all nodes and single token communication along all arcs. A considerable amount of theory was developed in this thesis (in chapter 3) to generate more accurate predictive models of communication and processing. Another difference is in the area of exploiting the tradeoff between parallelism and communication cost. The allocator in this thesis attempts to make this tradeoff systematically based on the separate *communication cost* and *processor multiplexing cost* components of the cost function. Program fragments that produce large amounts of communication delay relative to the amount of parallelism exposed are allocated to the same processor. In the extreme, the entire program may get allocated to the same processor even when more processors are available. The DDM2 allocator will expose all concurrency if there are sufficient numbers of processors available.

4.4.3.2 Hughes Dataflow Multiprocessor

Michael Campbell [10] describes another method for the compile-time allocation of dataflow programs to the Hughes Dataflow Multiprocessor. The multiprocessor has a bussed cube interconnection network. However, the allocation algorithms are not designed to work with just that topology.

Allocation is based on a heuristic cost function that is a weighted sum of a

communication cost and processing cost. Communication cost associated with the allocation of a single node in the dataflow graph is the sum of the distances of arcs connected with the node; distance is simply the number of hops from the processor associated with the source node of an arc to the processor associated with the destination node of the arc. No consideration is given to the size of the data in each token transmitted along an arc or the number of tokens. The processing cost is computed by first finding the transitive closure of the graph. Potentially parallel nodes are those that do not have an arc connecting them in the transitive closure. The processing cost associated with the allocation of a certain node to a processor is computed from the number of potentially parallel nodes allocated to the same processor. Each node is assumed to take the same computation time and no special consideration is given to the multiple invocation of a node.

The differences from this thesis are the following: (1) A much simpler model of communication is assumed here. (2) A much simpler model of processing is assumed. (3) Potentially parallel computations are found by computing the transitive closure of the graph. The allocator in this thesis performs an abstract simulation with probabilistic analysis to find parallel computations. (4) A node may be allocated to a single processor only. We allow multiple copies.

4.4.3.3 Vivek Sarkar's thesis

In his thesis *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors* [55], Vivek Sarkar describes another approach to compile-time allocation for dataflow programs. This approach is interesting because it takes completion time as the cost function as opposed to a combination of communication and processing. Some differences from this thesis are described below. First, it assumes that each processor has sufficient memory to execute the entire program unlike the approach in this thesis. Second, profile information is used for estimates as opposed to probabilistic estimates in this thesis. Finally, it is claimed that the approach is applicable to topologies in which there could be delays that are a function of the distance between processors. However, all experiments reported assume delays that are independent of distance.

4.4.4 Kemal Oflazer's Thesis on Partitioning of Production Systems

Kemal Oflazer discusses the partitioning problem for Production Systems (or Rule-Based Systems), specifically *OPS5* [23], in his thesis *Partitioning in Parallel Processing of Production Systems* [50] and an earlier paper [49]. This partitioning problem is described as the compile-time allocation of productions (or rules) to processors in such a way that the total time of execution is minimized.

A production system interpreter repeatedly executes a *recognize-act* cycle. This cycle consists of 3 phases—*Match*, *Conflict-Resolution*, and *Act*. The *Match* phase finds all productions that may be fired, the *Conflict-Resolution* phase picks a single production to be fired, and the *Act* phase performs the changes to the database mandated by the chosen production. Note that the *Conflict-Resolution* phase is a synchronization point during every cycle.

In Oflazer's parallel processor organization for partitioning, a set of processors contains mutually exclusive and exhaustive subsets of the productions in the systems. Each processor also contains the state associated with its subset of the productions. The goal of each processor is to make any changes to its state mandated by the previous *Act* phase, find the matching productions, and report them to some central processing location. The central processor performs the *Conflict Resolution phase* and identifies the state changes mandated by the chosen production to the relevant processors. Since most of the processing in production systems takes place during the *Match* phase, Oflazer's model ignores the processing cost during the *Conflict-Resolution* phase and the *Act* phase. In addition, communication cost between the parallel processors and the central processing location is ignored because it is a small amount of data.

This work is different from our model in the following ways. First, the presence of a synchronization point during every interpreter cycle makes it a very different type of computation. There are no such synchronization points in dataflow* graphs. Second, communication cost is not a factor in Oflazer's work whereas it is a central focus of the work in this thesis. Third, Oflazer takes the estimates for processing

costs from previous executions of the same production system. In our case, estimates are produced by probabilistic analysis.

4.4.5 Compile-time Allocation of Actor Languages

Bill Athas has recently completed a thesis on compile-time allocation for a concurrent, object-oriented programming language called *Cantor* [4]. Unfortunately, the thesis was not available in time to make a detailed comparison.

4.4.6 Run-time Allocation

A lot of research has been done in the area of run-time allocation for many different types of computations. It is not possible to discuss all the work here but some interesting pieces of work are mentioned below. As mentioned earlier in chapter 1, run-time allocation has the disadvantage that the overhead of decision-making must be paid at run-time. However, if the behavior of the program is highly dynamic and is hard to predict at compile-time, then run-time allocation may be the best approach.

Smith [66] has presented a protocol called *Contract Net* to dynamically distribute tasks among processors in a distributed system. Each task is distributed using an *Announcement-Bid-Award* sequence. A task to be distributed is *announced* as being available, processors may *bid* to do the task, and the announcing processor may then *award* the *contract* to one of the processors. The idea was to propose a more flexible framework than some other rigid frameworks like *remote procedure calls* [47], for example.

Malone et. al [42] have proposed an interesting specialization of the Contract Net (called *Enterprise*) and showed some good connections to scheduling theory results. Singh and Genesereth [61] proposed another specialization of the *Contract Net* (called *Variable Supply Model*) that was shown to be an efficient and flexible approach to distributing or-parallel tasks on a broadcast network.

Hornig [36] has designed a distributed reduction-style interpreter for a functional

language he designed called *Stardust*. An interesting feature of this work is that user-defined functions are annotated with time estimates provided by the user. Time estimates can be arbitrary functions of the arguments to the function. Several examples were presented in which time estimates can be provided reasonably. Such time estimates could be useful for compile-time allocation as well.

Haridi and Ciepielewski [33] have described a *token-pool* mechanism to distribute or-parallel logic programs. The idea is that or-parallel computations are encapsulated in *tokens*. These tokens may be placed in the pool as they are generated and picked up by other processors. The difference from the Contract Net is that computations are not handed over directly from the spawning processor to the contracting processor. The token pool acts as an intermediary between the spawning processor and the contracting processor. However, the token pool seems to be a passive entity. Therefore, the spawning processor does not have any control over which contracting processor gets selected for any spawned computation.

Hermenegildo [34] and some others have provided an interesting twist to this idea of the token pool. The idea is that computations that can be spawned off to remote processors are simply kept in local storage at some well-known location. Remote processors can retrieve these parallel computations completely independently without any intervention of the local processor. Some special hardware may be needed for this mechanism but it has the advantage that the busy processors do not have to pay the overhead of distribution. It is the idle processors that must spend some time searching for some parallel computations to start working on.

4.4.7 Programmed Allocation

Shapiro [58] has described a notation for programmers to specify their own allocations for Concurrent Prolog programs [57]. The notation is based on the turtle notation of LOGO programs [52] and is very elegant. However, the programmer must have a very good idea of the structure of the program to make use of it. In cases where the dynamic behavior of the program is not well-known by the user, user-specified allocations are not likely to perform well.

4.5 Conclusions

This chapter contained the description of a compile-time allocation strategy based on a cost function that is not specific to any particular domain or multiprocessor. It was shown that the algorithms involved are tractable (i.e., they have polynomial worst-case time complexity). For the 4-bit adder example, this allocation strategy produced speedups that were more than half an unreachable upper bound. In the FAIM-1 configuration considered, communication costs are not high; therefore, even a random allocation does quite well (though not as well as greedy allocation). In general, it is possible that random allocations may perform arbitrarily worse than the allocation strategy presented here.

Chapter 5

Conclusions

5.1 Summary of Key Ideas

In this thesis, we presented solutions to two problems: (1) the design of a parallel execution model for backward-chaining deductions and (2) the allocation of the resulting parallel computations to an interesting class of multiprocessors.

The target class of multiprocessors has the following properties: (1) there are an arbitrary number of MIMD processors; (2) each processor has some local memory but there is no global memory; (3) processors can communicate only by sending messages to each other; (4) message delay is a function of the amount of data in the message and the distance between source and destination; and (5) each processor can perform backward-chaining deductions based on the subset of the program that it contains.

PM, the parallel execution model described in chapter 2, exploits more parallelism than other execution models that use data-driven control and the same target class of multiprocessors. In particular, *PM* exploits *or-parallelism*, *and-parallelism* and *pipelining*. The extra parallelism can be an important advantage in a situation where a large number of processors are available. Data-driven control leads to minimal synchronization overhead and means that the inherent parallelism can be fully exploited. The chapter included a correctness theorem that stated that the

set of solutions produced by *PM* is identical to the set of solutions produced by a Prolog interpreter. *PM* does not assume that the entire program can be stored in each processor's local memory. Therefore, larger programs can be run compared to the case in which a copy of the entire program is required in each processor's local memory.

We described a compile-time allocation strategy for *PM* in chapters 3 and 4. In order to compare different allocations, the strategy uses a cost function (described in chapter 3) that applies to any application and multiprocessor (in the target multiprocessor class). The cost function attempts to capture intuitive notions of the quality of allocations. The completion time of the computation, assuming zero computation delays and infinite processors, is the completion time for the associated parallelism profile. The non-zero delays and sequentialization of parallel computation associated with a realistic multiprocessor will increase this completion time. The cost function is defined to be an upper bound on this additional delay assuming that the effects of non-zero communication delays and sequentialization of parallel computation (due to a finite number of processors) are independent and, therefore, additive. The upper bound on the extra delay due to non-zero communication is given by the sum of all communication delays. This is called the communication cost of the computation. The upper bound on the extra delay due to sequentialization of parallel computation is called the processor multiplexing cost. The overall cost is the sum of the communication cost and the processor multiplexing cost.

An important feature of this cost function is that it can be efficiently computed and recomputed (for small changes in the allocation). Algorithms were presented for this computation and recomputation. Unfortunately, the algorithms require certain restrictions that *PM* does not require. First, the type of backward-chaining deduction is restricted. In particular, no recursive clauses are allowed, unit clauses must be ground, and certain probabilistic uniformity and independence assumptions must apply. Second, a partitioning of the database is assumed to be given.

Some of the probabilistic techniques used in the cost computation algorithms should be useful in other contexts as well. A couple of examples are given below. First, the *Communication Estimation* algorithm computes the expected amount of

communication between each pair of partitions. Since the trade-off between communication and parallelism seems to be so fundamental for the allocation problem, estimating communication should be useful for other allocation strategies. Second, computing the parallelism profile is a side-effect of the processor multiplexing cost computation. Parallelism profiles have been used for allocation strategies other than the one described here [55]. In addition, they are used sometimes simply for the purpose of estimating the amount of parallelism inherent in an application.

In chapter 4, we described a search strategy for finding a satisfactory allocation in the space of possible allocations. The search strategy consisted of a greedy allocation phase followed by a local minimization phase. Greedy allocation allocates partitions of the database to processors one at a time. A partition is allocated to the lowest cost processor without re-allocating any partitions that were allocated previously. The local minimization phase consists of a sequence of cost-reducing re-allocations of partitions to neighboring processors till a local minimum is reached. It was shown that both greedy allocation and each *round* of local minimization have worst-case time complexities that are polynomial.

Experiments indicate that greedy allocation alone produces quite satisfactory answers. For the 4-bit digital adder example that was tried on a simulation of the FAIM-1 multiprocessor, the speedup achieved by using the greedy allocation was more than half of an unreachable upper bound. Also, the speedup achieved was somewhat better than that achieved by using random allocation. More analysis revealed that random allocation works so well because this particular example is not communication intensive at all. There are at least two cases where the difference in performance between the allocation strategy advocated and the random allocation strategy can be expected to be significant. First, a higher number of processors will increase the average distance and, therefore, the average delay for the random allocation case. However, average distances need not increase at all for the allocation strategy advocated when more processors are used. Second, higher communication constants associated with a different multiprocessor with less communication support can cause the speedup to be arbitrarily close to zero. However, the allocation strategy advocated here will allocate all computation to a single processor (with a

speedup of 1) when communication cost overwhelms processor multiplexing cost.

5.2 Directions for Future Research

Two versions of the greedy allocation algorithm were described in chapter 4—*GreedyAllocation'* and *GreedyAllocation''*. However, experiments were conducted only with *GreedyAllocation''*. It is quite possible that *GreedyAllocation'* will lead to better allocations since the number of multiple copies is chosen in a less arbitrary manner than in *GreedyAllocation''*. The disadvantage of using *GreedyAllocation'* is that it has a higher time complexity.

A constraint that was kept in mind while designing the current cost function was to make recomputation efficient when small changes are made to the allocation. However, as the experiments indicate, greedy allocation by itself produced quite reasonable allocations without using local minimization at all. Since recomputation is useful only for local minimization, there is the possibility now of using a different cost function that is not as pessimistic as the current cost function and one that is not necessarily designed for efficient recomputation. A more accurate cost function of this type has the potential of improving the quality of the greedy allocation algorithm.

At present, the allocation techniques do not apply to recursive cases. If arbitrary recursions are allowed, it becomes undecidable to predict the amount of processing and communication required for a parallel computation.[1] Therefore, good allocation decisions are unlikely. However, it may be possible to reason automatically about restricted recursive cases. Even in cases where completely automatic allocation is not possible, users may provide information about parallel computation and communication to make reasonable allocation decisions possible.

In many Artificial Intelligence problems, a single solution is required for the problem at hand. It should be possible to extend *PM* to kill off redundant processes when the first solution has been found. It may be harder to extend the allocation techniques to reason about the modified parallel execution model. On a related

[1] This follows directly from the halting problem.

issue, *PM* should be modified to kill off processes associated with sibling and-nodes when no solution is found for any one of the and-nodes.

Over the years, researchers have developed compilation techniques for Prolog that make it execute at comparable speeds with other programming languages for comparable problems [73,72]. More attention should be directed towards applying this compilation technology, perhaps with extensions, to parallel execution models like *PM*.

Although backward-chaining deduction has been found to be very useful for a wide range of problems, other types of deduction are more natural for certain applications. For example, simulation is better done with forward-chaining deduction [60] and planning problems are better handled with Residue [21]. Techniques for exposing the parallelism in these types of deduction will be needed if the associated applications are to be speeded up.

The allocation techniques described in this thesis were directed towards Horn clause databases without any additional annotations. In the literature, this is called the implicit parallelism case for logic programming in contrast to logic programming languages that require explicit annotations to express producer-consumer relationships between processes. Explicitly parallel logic programming languages include Concurrent Prolog [57], PARLOG [30], and Guarded Horn Clauses (GHC) [71]. The extent to which the allocation techniques in this thesis are applicable to these languages remains to be seen. Going even further, the applicability of the allocation techniques to other programming paradigms like object-oriented languages (e.g., Actors [1]) and Lisp-based languages (e.g., Qlisp [26] and Multilisp [31,32]) should be investigated.

As mentioned earlier, compile-time allocation works best when good estimates can be made at compile-time about run-time program behavior. If good compile-time predictions can be made for some parts of the program and not for others, it may make sense to use a hybrid strategy using both compile-time and run-time allocation. A hybrid strategy may also include some user-specified allocations when the user already knows how to allocate a piece of the computation exceptionally well.

Appendix A

Partial Order Algorithm

This algorithm describes how to pick a partial order for a conjunctive goal. In particular, the partial order is represented by a directed, acyclic graph of nodes representing the conjuncts.

On invoking a rule in backward-chaining, the antecedents of the rule become a new conjunctive subgoal that the inference engine may try to prove. Assume that appropriate bindings, resulting from the unification of the goal with the consequent of the rule, have been plugged into the antecedents.

A.1 Definitions

Let C_1 through C_n be the antecedents of the rule in order from left to right. Let CL be the ordered set of the antecedents of the rule.

$$CL =< C_1, C_2, \ldots, C_n >$$

The function v is defined to take a literal as argument and return the set of variables in the literal. For example,

$$v(p(X, Y, c)) = \{X, Y\}$$

The function vl is defined to take an ordered set of literals and return the set of variables in the literals.

$$vl(< C_1, C_2, \ldots, C_n >) = \bigcup_{i=1}^{n} v(C_i)$$

For example,

$$vl(< p(X, Y, c1), q(Y, Z, c2) >) = \{X, Y, Z\}$$

Let $d(C_i, C_j)$ be true if and only if there is a directed arc between the corresponding nodes in the *conjunct graph.*

As described in chapter 2, *PM* allows conjuncts to be solved in parallel only if previously solved conjuncts have already bound any shared variables that they may have. Let us call this constraint the *shared-variable constraint.* Restating the constraint, a single conjunct must first bind any given variable in $vl(CL)$, where CL is the ordered set of conjuncts, before other conjuncts that share the same variable can be solved. This distinguished conjunct is called the *generator conjunct* for the variable in question. Let $g(V, C_i)$ be true if and only if C_i is the *generator conjunct* of the variable V.

A.2 Assumption

No assertions (i.e., unit clauses in a horn clause database) contain any variables.

A.3 Algorithm

Input: CL, an ordered set of conjuncts

Output: A conjunct graph (i.e., a set of directed arcs between the conjuncts) such that (1) the partial order represented by the conjunct graph is a subset of the total order given in the input, (2) the partial order is the minimal one satisfying condition (1) and the *shared-variable constraint*, and (3) the conjunct graph is a minimal representation of the partial order. The term "minimal" is used with reference to the number of edges.

Condition (1) is chosen because it is expected that if the original total order is an efficient one, then subsets of it are also efficient. Condition (2) is chosen so that parallelism is maximized. Condition (3) is chosen so that communication requirements for *PM* are minimized. Reduced communication also translates into reduced computation at the nodes where the communication is directed.

There are three parts of the algorithm and these are now described one by one.

The first part of the algorithm picks a generator conjunct for each variable. For each variable in $vl(CL)$, pick the leftmost conjunct, C_i, in CL, such that the variable is contained in $v(C_i)$. This conjunct is declared to be the generator of the variable in question. The complexity of this part of the algorithm is $O(n \times k)$, where n is the number of conjuncts and k is the number of variables.

This can be illustrated with an example. Consider the conjunctive goal

$$p(X) \wedge q(Y) \wedge s(X,Y)$$

In this case,

$$\begin{aligned} CL &= < C_1, C_2, C_3 > \\ C_1 &= p(X) \\ C_2 &= q(Y) \\ C_3 &= s(X,Y) \\ vl(CL) &= \{X,Y\} \\ v(C_1) &= \{X\} \\ v(C_2) &= \{Y\} \\ v(C_3) &= \{X,Y\} \end{aligned}$$

The generator conjuncts are described by $g(X,C_1)$ and $g(Y,C_2)$.

In the second part of the algorithm, directed arcs are introduced between the generator conjuncts and other conjuncts. For each generator conjunct and each other conjunct that contains the variable *generated* by the generator, insert a directed arc between the corresponding nodes in the partial order graph. The complexity of this is $O(n^2)$, where n is the number of conjuncts in CL. Again, this is best illustrated

by an example. Consider the same example that was just considered above for picking the generator conjuncts. Since the generator conjunct for variable X is C_1 (i.e., p(X)) and it is the case that C_3 (i.e., s(X,Y)) contains the same variable, a directed arc, $d(C_1, C_3)$, is introduced. Similar reasoning leads to the only other directed arc $d(C_2, C_3)$. At this point, the partial order described by the set of directed arcs satisfies the *shared-variable constraint.* However, this may not be a minimal partial order satisfying the constraint as shown in a different example below.

It is possible that the partial order generated by the algorithm so far is as given below:

$$\{d(C_1, C_2), d(C_2, C_3), d(C_1, C_3)\}$$

This would happen if the variables contained in C_1, C_2, and C_3 are $\{X\}$, $\{X, Y\}$, and $\{X, Y, Z\}$ respectively. The arc $d(C_1, C_3)$ represents a redundant arc and can be removed while still maintaining the *shared-variable constraint.* Such arcs are called transitive arcs. An arc is a transitive arc if and only if there is a longer path between the end nodes of the arc.

A paper by Aho, Garey, and Ullman [2] shows how to remove all these transitive arcs from a directed, acyclic graph in time $O(n^3)$, where n is the number of vertices. The output of the algorithm is called the *transitive reduction* of the input graph. This transitive reduction algorithm is the third part of the partial order algorithm.

The overall complexity of the partial order algorithm is obtained by adding the complexities of the three component procedures. The complexity is $O(n \times k + n^2 + n^3)$ or $O(n^3)$, assuming that k is $O(n^3)$.

A.4 Another Example

Consider the rule

$$color(A, B, C, D, E) :-$$
$$next(A, B) \wedge next(C, D) \wedge next(A, C) \wedge next(A, D) \wedge$$
$$next(B, C) \wedge next(B, E) \wedge next(C, E) \wedge next(D, E)$$

This rule is part of the database used for a particular instance of the four color problem. A goal

$$color(A, B, C, D, E)$$

would generate the conjunctive subgoal

$$next(A, B) \wedge next(C, D) \wedge next(A, C) \wedge next(A, D) \wedge$$
$$next(B, C) \wedge next(B, E) \wedge next(C, E) \wedge next(D, E)$$

Now, $next(A, B)$ is the generator for both A and B. Also, $next(C, D)$ is the generator for both C and D. Finally, $next(B, E)$ is the generator for E.

The partial order contains the following directed arcs: From $next(A, B)$ to each member of $\{next(A, C), next(A, D), next(B, C), next(B, E)\}$, from $next(C, D)$ to each member of $\{next(A, C), next(A, D), next(B, C), next(C, E), next(D, E)\}$, and from $next(B, E)$ to each member of $\{next(C, E), next(D, E)\}$.

There are no transitive arcs to remove in this case.

Appendix B

Details of Procedure PMCCA-1

The procedure PMCCA-1 is used to compute processor multiplexing cost for a single processor. Chapter 3 described this procedure but omitted details of the two procedures *InsertPI* and *DeletePI*. These procedures are given in this appendix in more detail (in sections B.2 and B.3). The procedures use the abstract data structure called *PQ-list* and its description is repeated in section B.1 for the reader's convenience.

Pseudo-Pascal code is given for the procedures, with comments being delimited by "/*" on the left and "*/" on the right.

B.1 PQ-list Data Structure

This abstract data structure has three associated abstract operations as described in chapter 3.

1. *InsertPQL(PQL, element, key)*: This inserts the element *element* into the PQ-list *PQL* in log time. In addition, the *CLoad* field of the element is set to the *CLoad* field of the previous element (in sorted order). If there is no previous element, then the field is set to zero.

2. *DeletePQL(PQL, element)*: This deletes *element* from *PQL* in log time.

3. *EnumeratePQL(PQL, element1, element2)*: Enumerates all elements in PQL in sorted order from the element *element1* to the element *element2*. This is done in time linear in the number of elements enumerated.

The list is doubly-linked to allow forward or backward traversal. The utility of backward pointers will become apparent later in the description of procedures *InsertPI* and *DeletePI*.

B.2 Procedure *InsertPI*

The *InsertPI* procedure is given in figure 36.

The insert procedure uses *InsertPQL* to insert the two end-points of the processing intervals as two elements into the *PQ-list*. It also enumerates all the elements from the start element to the finish element and modifies their *CLoad* appropriately to reflect the change. In addition, PMC is changed as each element is considered. The correct PMC is available at the end of the procedure.

B.3 Procedure *DeletePI*

The *DeletePI* procedure is given in figure 37.

The delete procedure enumerates all the elements from the start element to the finish element. It modifies the *CLoad* values of the elements appropriately. Also, PMC associated with the *PQ-list* is changed as each element is considered. The correct PMC is available at the end of the procedure. Moreover, the start and finish elements are deleted from the data-structure (using *DeletePQL*) when they are enumerated.

```
Procedure InsertPI(PI, PQList);
    begin
        HI ← PI.ProcessorLoad;
        /* PMC =Current value of processor multiplexing cost */
        StartElem ← InsertPQL(start(PI), start(PI).Time);
        FinishElem ← InsertPQL(finish(PI), finish(PI).Time);
        If StartElem.Prev ≠ nil then begin
        /* The prev field is the previous element in sorted order. */
            CLoadPrev ← StartElem.Prev.CLoad;
            CLoadPrevOld ← StartElem.Prev.CLoad;
            TimePrev ← StartElem.Prev.Time
        end
        else begin
            CLoadPrev ← 0;
            CLoadPrevOld ← 0;
            TimePrev ← 0
        end;
        For all Elem ∈ EnumeratePQL( PQList, StartElem, FinishElem) do begin
            PMC ← PMC+ [max(0, CLoadPrev − 1) − max(0, CLoadPrevOld − 1)]×
                                (Elem.Time − TimePrev);
            CLoadPrevOld ← Elem.CLoad;
            If Elem ≠ FinishElem then
               Elem.CLoad ← Elem.CLoad + HI;
            CLoadPrev ← Elem.CLoad;
            TimePrev ← Elem.Time
        end /* for */
    end; /* InsertPI */
```

Figure 36: Procedure *InsertPI*

```
Procedure DeletePI(Elem1, Elem2, PQList);
    begin
        HI ← PI.ProcessorLoad;
        /* PMC =Current value of processor multiplexing cost */
        If Elem1.Prev.CLoad ≠ nil then begin
            CLoadPrev ← Elem1.Prev.CLoad;
            CLoadPrevOld ← Elem1.Prev.CLoad;
            TimePrev ← Elem1.Prev.Time
        end
        else begin
            CLoadPrev ← 0;
            CLoadPrevOld ← 0;
            TimePrev ← 0
        end;
        For all Elem ∈ EnumeratePQL( PQList, Elem1, Elem2) do begin
            PMC ← PMC+ [max(0, CLoadPrev − 1) − max(0, CLoadPrevOld − 1)]×
                        (Elem.Time − TimePrev);
            CLoadPrevOld = Elem.CLoad;
            If Elem ≠ Elem2 then
                Elem.CLoad = Elem.CLoad − HI;
            CLoadPrev = Elem.CLoad;
            If Elem = Elem1 or Elem = Elem2 then
                DeletePQL(PQList, Elem);
            TimePrev ← Elem.Time;
        end /* for */
    end; /* DeletePI */
```

Figure 37: Procedure *DeletePI*

Appendix C

Greedy Allocation is not Locally Optimal

This appendix presents an example where greedy allocation does not produce a locally optimal solution. Consider the dataflow* graph in figure 38 and the processor topology shown in figure 39. Assume that there is no processing overlap between nodes A or B with C. Also, let there be no overlap between nodes B or C with D. Let the amounts of communication between the node pairs A and B and separately A and C be very low and equal to each other. Also, let the amounts of communication between the node pairs B and D and separately C and D be very high and equal to each other. If the greedy allocation algorithm allocates the nodes in the topological order A, B, C, and then D, a possible allocation may be as given below:

$$A \longrightarrow 1$$

$$B \longrightarrow 1$$

$$C \longrightarrow 2$$

$$D \longrightarrow 1$$

When B and C get allocated by the greedy allocation procedure, the only communication considered is from A to B and C. However, this is not necessarily the

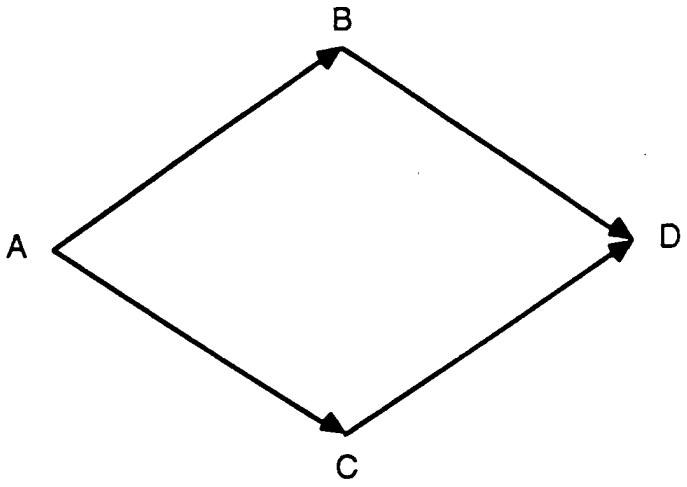


Figure 38: A Dataflow* Graph

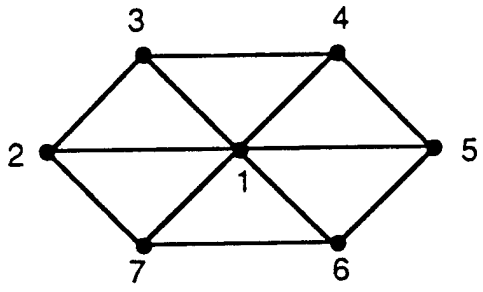


Figure 39: A Processor Topology

locally optimal solution. If communication from B and C to D is high enough relative to the communication from A to B and C, then it may reduce the cost function by moving C from processor 2 to 1. Although the processor multiplexing cost is increased, the effect due to reduction of communication cost may be greater.

Appendix D

Local Minimization is not Globally Optimal

This appendix presents an example where local minimization of an allocation does not produce a globally optimal solution. Consider the dataflow* graph in figure 40 and the processor topology shown in figure 41. Assume that there is no processing overlap between node A with any of the nodes in the set {B, C, D, E}. Also, assume that there is no overlap between any of the nodes in the set {B, C, D, E} with node F. Let the amounts of communication from A with any node in the set {B, C, D, E} be equal and very low. Let the amounts of communication from any node in {B, C, D, E} with node F be equal and very large. Assume, in addition, that memory requirements dictate that at most one node may be allocated to a single processor. If the greedy allocation algorithm allocates nodes in the topological order A, B, C, D, E, and then F, a possible allocation may be as given below:

$$A \longrightarrow 1$$

$$B \longrightarrow 2$$

$$C \longrightarrow 4$$

$$D \longrightarrow 6$$

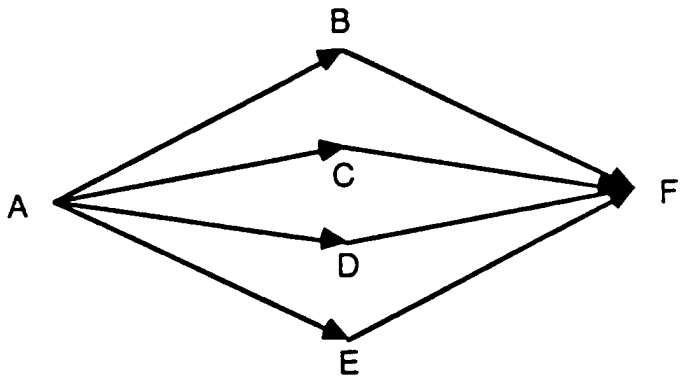


Figure 40: A Dataflow* Graph

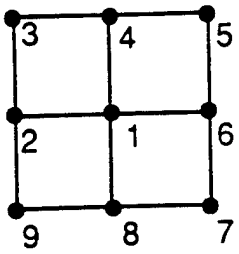


Figure 41: A Processor Topology

$$E \longrightarrow 8$$

$$F \longrightarrow 3$$

The allocation produced by the greedy allocation procedure is already a locally optimal solution because any feasible local neighbor has a higher cost. However, it is easy to see that this not necessarily the globally optimal solution. Given that communication from nodes in the set {B, C, D, E} to F is high enough compared to the communication from A to the nodes in the set {B, C, D, E}, then a lower cost solution is given below:

$$F \longrightarrow 1$$

$$B \longrightarrow 2$$

$$C \longrightarrow 4$$

$$D \longrightarrow 6$$

$$E \longrightarrow 8$$

$$A \longrightarrow 3$$

It is interesting to note that if the order chosen for greedy allocation had been reversed, this lower cost allocation would have been the one generated.

Examples can also be generated that are locally optimal but not globally optimal and where the size of memory is not an issue.

Appendix E

Adder Example

E.1 Syntax and Notation

Anything followed by ";" on a line is a comment. The syntax for facts and rules in MRS is different from the standard Prolog syntax. Variables are symbols that begin with the character "$". A literal in Prolog as in
<predicate>(<field1>,<field2>,...,<fieldn>)
is written in MRS as
(<predicate> <field1> <field2> ... <fieldn>).
A rule in Prolog as in
<goal> :- <subgoal1>,<subgoal2>,...,<subgoaln>
is written in MRS as
(if (and <subgoal1> <subgoal2> ... <subgoaln>) <goal>).

In addition, a fact at compile-time is represented in a different way than at run-time. Compile-time facts are written as
(fact <run-time-fact> <list-of-variables> <number>).

<list-of-variables> indicates that all the variables in the list are actually unknown constants in the run-time fact <run-time-fact>. <number> is the number of facts matching this fact pattern that are expected to be present at run-time.

The device whose structure and behavior is captured here is called "FOO". It

is a 4-bit adder. The database contains many literals of the form
(VALUE (PORT <port-name> <device>) <value>).

This is intended to mean that the value of the specified port <port> of device <device> is <value>. <device> is either the top-level device "FOO" or parts of it specified in a hierarchical fashion. (PART (NUM FA i) FOO) for i from 1 to 4 stands for the ith 1-bit full adder. Each 1-bit full adder is composed of 5 gates: 1 or gate, 2 exclusive-or gates, and 2 and gates. An example of a device at this lowest level is (PART OR1 (PART (NUM FA 4.) FOO)) This represents the 1st or gate (OR1) of the 4th 1-bit full adder (FA) of the top-level device (FOO).

E.2 Adder Database

E.2.1 Adder Database at Run-Time

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; External inputs at run-time

(VALUE (PORT IN1 (PART (NUM FA 1.) FOO)) 1.)

(VALUE (PORT IN1 (PART (NUM FA 2.) FOO)) 1.)

(VALUE (PORT IN1 (PART (NUM FA 3.) FOO)) 1.)

(VALUE (PORT IN1 (PART (NUM FA 4.) FOO)) 1.)

(VALUE (PORT IN2 (PART (NUM FA 1.) FOO)) 1.)

(VALUE (PORT IN2 (PART (NUM FA 2.) FOO)) 0.)

(VALUE (PORT IN2 (PART (NUM FA 3.) FOO)) 0.)

(VALUE (PORT IN2 (PART (NUM FA 4.) FOO)) 0.)

(VALUE (PORT CIN (PART (NUM FA 1.) FOO)) 0.)
```

```
;;; End of external inputs
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(IF (VALUE (PORT OUT (PART OR1 (PART (NUM FA 1.) FOO))) $358.)
    (VALUE (PORT COUT (PART (NUM FA 1.) FOO)) $358.))

(IF (VALUE (PORT OUT (PART OR1 (PART (NUM FA 2.) FOO))) $356.)
    (VALUE (PORT COUT (PART (NUM FA 2.) FOO)) $356.))

(IF (VALUE (PORT OUT (PART OR1 (PART (NUM FA 3.) FOO))) $354.)
    (VALUE (PORT COUT (PART (NUM FA 3.) FOO)) $354.))

(IF (VALUE (PORT OUT (PART OR1 (PART (NUM FA 4.) FOO))) $352.)
    (VALUE (PORT COUT (PART (NUM FA 4.) FOO)) $352.))

(IF (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 1.) FOO))) $350.)
    (VALUE (PORT SUM (PART (NUM FA 1.) FOO)) $350.))

(IF (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 2.) FOO))) $348.)
    (VALUE (PORT SUM (PART (NUM FA 2.) FOO)) $348.))

(IF (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 3.) FOO))) $346.)
    (VALUE (PORT SUM (PART (NUM FA 3.) FOO)) $346.))

(IF (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 4.) FOO))) $344.)
    (VALUE (PORT SUM (PART (NUM FA 4.) FOO)) $344.))

(IF (VALUE (PORT CIN (PART (NUM FA 1.) FOO)) $342.)
    (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 1.) FOO))) $342.))

(IF (VALUE (PORT CIN (PART (NUM FA 2.) FOO)) $340.)
    (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 2.) FOO))) $340.))

(IF (VALUE (PORT CIN (PART (NUM FA 3.) FOO)) $338.)
    (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 3.) FOO))) $338.))

(IF (VALUE (PORT CIN (PART (NUM FA 4.) FOO)) $336.)
    (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 4.) FOO))) $336.))
```

```
(IF (VALUE (PORT CIN (PART (NUM FA 1.) FOO)) $334.)
    (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 1.) FOO))) $334.))

(IF (VALUE (PORT CIN (PART (NUM FA 2.) FOO)) $332.)
    (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 2.) FOO))) $332.))

(IF (VALUE (PORT CIN (PART (NUM FA 3.) FOO)) $330.)
    (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 3.) FOO))) $330.))

(IF (VALUE (PORT CIN (PART (NUM FA 4.) FOO)) $328.)
    (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 4.) FOO))) $328.))

(IF (VALUE (PORT IN2 (PART (NUM FA 1.) FOO)) $326.)
    (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 1.) FOO))) $326.))

(IF (VALUE (PORT IN2 (PART (NUM FA 2.) FOO)) $324.)
    (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 2.) FOO))) $324.))

(IF (VALUE (PORT IN2 (PART (NUM FA 3.) FOO)) $322.)
    (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 3.) FOO))) $322.))

(IF (VALUE (PORT IN2 (PART (NUM FA 4.) FOO)) $320.)
    (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 4.) FOO))) $320.))

(IF (VALUE (PORT IN2 (PART (NUM FA 1.) FOO)) $318.)
    (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 1.) FOO))) $318.))

(IF (VALUE (PORT IN2 (PART (NUM FA 2.) FOO)) $316.)
    (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 2.) FOO))) $316.))

(IF (VALUE (PORT IN2 (PART (NUM FA 3.) FOO)) $314.)
    (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 3.) FOO))) $314.))

(IF (VALUE (PORT IN2 (PART (NUM FA 4.) FOO)) $312.)
    (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 4.) FOO))) $312.))

(IF (VALUE (PORT IN1 (PART (NUM FA 1.) FOO)) $310.)
```

```
    (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 1.) FOO))) $310.))

(IF (VALUE (PORT IN1 (PART (NUM FA 2.) FOO)) $308.)
    (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 2.) FOO))) $308.))

(IF (VALUE (PORT IN1 (PART (NUM FA 3.) FOO)) $306.)
    (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 3.) FOO))) $306.))

(IF (VALUE (PORT IN1 (PART (NUM FA 4.) FOO)) $304.)
    (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 4.) FOO))) $304.))

(IF (VALUE (PORT IN1 (PART (NUM FA 1.) FOO)) $302.)
    (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 1.) FOO))) $302.))

(IF (VALUE (PORT IN1 (PART (NUM FA 2.) FOO)) $300.)
    (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 2.) FOO))) $300.))

(IF (VALUE (PORT IN1 (PART (NUM FA 3.) FOO)) $298.)
    (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 3.) FOO))) $298.))

(IF (VALUE (PORT IN1 (PART (NUM FA 4.) FOO)) $296.)
    (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 4.) FOO))) $296.))

(IF (VALUE (PORT COUT (PART (NUM FA 1.) FOO)) $275.)
    (VALUE (PORT CIN (PART (NUM FA 2.) FOO)) $275.))

(IF (VALUE (PORT COUT (PART (NUM FA 2.) FOO)) $273.)
    (VALUE (PORT CIN (PART (NUM FA 3.) FOO)) $273.))

(IF (VALUE (PORT COUT (PART (NUM FA 3.) FOO)) $271.)
    (VALUE (PORT CIN (PART (NUM FA 4.) FOO)) $271.))

(IF (VALUE (PORT OUT (PART AND2 (PART (NUM FA 1.) FOO))) $269.)
    (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 1.) FOO))) $269.))

(IF (VALUE (PORT OUT (PART AND1 (PART (NUM FA 1.) FOO))) $267.)
    (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 1.) FOO))) $267.))
```

```
(IF (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 1.) FOO))) $265.)
    (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 1.) FOO))) $265.))

(IF (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 1.) FOO))) $263.)
    (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 1.) FOO))) $263.))

(IF (AND (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 1.) FOO))) 1.)
         (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 1.) FOO))) $261.))
    (VALUE (PORT OUT (PART AND2 (PART (NUM FA 1.) FOO))) $261.))

(IF (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 1.) FOO))) 0.)
    (VALUE (PORT OUT (PART AND2 (PART (NUM FA 1.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 1.) FOO))) 1.)
         (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 1.) FOO))) $258.))
    (VALUE (PORT OUT (PART AND1 (PART (NUM FA 1.) FOO))) $258.))

(IF (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 1.) FOO))) 0.)
    (VALUE (PORT OUT (PART AND1 (PART (NUM FA 1.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 1.) FOO))) 0.)
         (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 1.) FOO))) $255.))
    (VALUE (PORT OUT (PART OR1 (PART (NUM FA 1.) FOO))) $255.))

(IF (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 1.) FOO))) 1.)
    (VALUE (PORT OUT (PART OR1 (PART (NUM FA 1.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 1.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 1.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 1.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 1.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 1.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 1.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 1.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 1.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 1.) FOO))) 1.))
```

```
(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 1.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 1.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 1.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 1.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 1.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 1.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 1.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 1.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 1.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 1.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 1.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 1.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 1.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 1.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 1.) FOO))) 0.))

(IF (VALUE (PORT OUT (PART AND2 (PART (NUM FA 2.) FOO))) $244.)
    (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 2.) FOO))) $244.))

(IF (VALUE (PORT OUT (PART AND1 (PART (NUM FA 2.) FOO))) $242.)
    (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 2.) FOO))) $242.))

(IF (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 2.) FOO))) $240.)
    (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 2.) FOO))) $240.))

(IF (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 2.) FOO))) $238.)
    (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 2.) FOO))) $238.))

(IF (AND (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 2.) FOO))) 1.)
         (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 2.) FOO))) $236.))
    (VALUE (PORT OUT (PART AND2 (PART (NUM FA 2.) FOO))) $236.))

(IF (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 2.) FOO))) 0.)
```

```
    (VALUE (PORT OUT (PART AND2 (PART (NUM FA 2.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 2.) FOO))) 1.)
         (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 2.) FOO))) $233.))
    (VALUE (PORT OUT (PART AND1 (PART (NUM FA 2.) FOO))) $233.))

(IF (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 2.) FOO))) 0.)
    (VALUE (PORT OUT (PART AND1 (PART (NUM FA 2.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 2.) FOO))) 0.)
         (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 2.) FOO))) $230.))
    (VALUE (PORT OUT (PART OR1 (PART (NUM FA 2.) FOO))) $230.))

(IF (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 2.) FOO))) 1.)
    (VALUE (PORT OUT (PART OR1 (PART (NUM FA 2.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 2.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 2.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 2.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 2.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 2.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 2.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 2.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 2.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 2.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 2.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 2.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 2.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 2.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 2.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 2.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 2.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 2.) FOO))) 1.))
```

```
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 2.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 2.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 2.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 2.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 2.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 2.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 2.) FOO))) 0.))

(IF (VALUE (PORT OUT (PART AND2 (PART (NUM FA 3.) FOO))) $219.)
    (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 3.) FOO))) $219.))

(IF (VALUE (PORT OUT (PART AND1 (PART (NUM FA 3.) FOO))) $217.)
    (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 3.) FOO))) $217.))

(IF (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 3.) FOO))) $215.)
    (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 3.) FOO))) $215.))

(IF (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 3.) FOO))) $213.)
    (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 3.) FOO))) $213.))

(IF (AND (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 3.) FOO))) 1.)
         (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 3.) FOO))) $211.))
    (VALUE (PORT OUT (PART AND2 (PART (NUM FA 3.) FOO))) $211.))

(IF (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 3.) FOO))) 0.)
    (VALUE (PORT OUT (PART AND2 (PART (NUM FA 3.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 3.) FOO))) 1.)
         (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 3.) FOO))) $208.))
    (VALUE (PORT OUT (PART AND1 (PART (NUM FA 3.) FOO))) $208.))

(IF (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 3.) FOO))) 0.)
    (VALUE (PORT OUT (PART AND1 (PART (NUM FA 3.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 3.) FOO))) 0.)
         (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 3.) FOO))) $205.))
```

```
    (VALUE (PORT OUT (PART OR1 (PART (NUM FA 3.) FOO))) $205.))

(IF (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 3.) FOO))) 1.)
    (VALUE (PORT OUT (PART OR1 (PART (NUM FA 3.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 3.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 3.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 3.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 3.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 3.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 3.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 3.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 3.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 3.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 3.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 3.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 3.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 3.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 3.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 3.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 3.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 3.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 3.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 3.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 3.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 3.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 3.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 3.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 3.) FOO))) 0.))

(IF (VALUE (PORT OUT (PART AND2 (PART (NUM FA 4.) FOO))) $194.)
```

```
    (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 4.) FOO))) $194.))

(IF (VALUE (PORT OUT (PART AND1 (PART (NUM FA 4.) FOO))) $192.)
    (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 4.) FOO))) $192.))

(IF (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 4.) FOO))) $190.)
    (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 4.) FOO))) $190.))

(IF (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 4.) FOO))) $188.)
    (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 4.) FOO))) $188.))

(IF (AND (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 4.) FOO))) 1.)
         (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 4.) FOO))) $186.))
    (VALUE (PORT OUT (PART AND2 (PART (NUM FA 4.) FOO))) $186.))

(IF (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 4.) FOO))) 0.)
    (VALUE (PORT OUT (PART AND2 (PART (NUM FA 4.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 4.) FOO))) 1.)
         (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 4.) FOO))) $183.))
    (VALUE (PORT OUT (PART AND1 (PART (NUM FA 4.) FOO))) $183.))

(IF (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 4.) FOO))) 0.)
    (VALUE (PORT OUT (PART AND1 (PART (NUM FA 4.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 4.) FOO))) 0.)
         (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 4.) FOO))) $180.))
    (VALUE (PORT OUT (PART OR1 (PART (NUM FA 4.) FOO))) $180.))

(IF (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 4.) FOO))) 1.)
    (VALUE (PORT OUT (PART OR1 (PART (NUM FA 4.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 4.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 4.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 4.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 4.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 4.) FOO))) 1.))
```

```
   (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 4.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 4.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 4.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 4.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 4.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 4.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 4.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 4.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 4.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 4.) FOO))) 0.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 4.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 4.) FOO))) 1.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 4.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 4.) FOO))) 1.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 4.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 4.) FOO))) 1.))

(IF (AND (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 4.) FOO))) 0.)
         (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 4.) FOO))) 0.))
    (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 4.) FOO))) 0.))
```

E.2.2 Adder Database at Compile-Time

The rules are the same as the rules in the run-time database and will not be repeated here. The facts are written in a different fashion and these are given below.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; External inputs at compile-time

(fact (VALUE (PORT IN1 (PART (NUM FA 1.) FOO)) $x) ($x) 1)

(fact (VALUE (PORT IN1 (PART (NUM FA 2.) FOO)) $x) ($x) 1)
```

```
(fact (VALUE (PORT IN1 (PART (NUM FA 3.) FOO)) $x) ($x) 1)

(fact (VALUE (PORT IN1 (PART (NUM FA 4.) FOO)) $x) ($x) 1)

(fact (VALUE (PORT IN2 (PART (NUM FA 1.) FOO)) $x) ($x) 1)

(fact (VALUE (PORT IN2 (PART (NUM FA 2.) FOO)) $x) ($x) 1)

(fact (VALUE (PORT IN2 (PART (NUM FA 3.) FOO)) $x) ($x) 1)

(fact (VALUE (PORT IN2 (PART (NUM FA 4.) FOO)) $x) ($x) 1)

(fact (VALUE (PORT CIN (PART (NUM FA 1.) FOO)) $x) ($x) 1)

;;; End of external inputs
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

E.3 Goal

E.3.1 Goal at Run-Time

The goal is to determine the value of the "COUT" port of the fourth full-adder in the top-level device "FOO". The fourth full-adder is associated with the highest order bit. The goal is given below.

(VALUE (PORT COUT (PART (NUM FA 4) FOO)) $X)

E.3.2 Goal at Compile-Time

The syntax for goals at compile-time is similar to that for facts at compile-time. "GOAL" is the predicate as opposed to "FACT". The first term is the fact and the next two terms are the list of unknown constants and the number of goals respectively. The goal is shown below.

(GOAL (VALUE (PORT COUT (PART (NUM FA 4.) FOO)) $X) NIL 1.)

E.4 Domain Information

The cardinalities of the domains of all variables is 2 because variables can be bound to either 0 or 1.

E.5 Partitioning Database

In this database, a fact of the form
(PARTITION <FACT-PATTERN>)
indicates that all facts that match the fact pattern <FACT-PATTERN> or all rules whose consequents (or heads) match the fact pattern are included in one partition. Variables are now all symbols that begin with "&". The notation for variables is different here because there are two types of variables in MRS. Variables that begin with "$" are *base-level* variables and variables that begin with "&" are *meta-level* variables. The base-level database describes the application of interest and the meta-level describes information about the base-level. The distinction is not terribly important here except that the partitioning database is better thought of as containing meta-level information. The partitioning database is shown below.

```
(PARTITION (VALUE (PORT IN1 (PART (NUM FA 1) FOO)) &X))
(PARTITION (VALUE (PORT IN1 (PART (NUM FA 2) FOO)) &X))
(PARTITION (VALUE (PORT IN1 (PART (NUM FA 3) FOO)) &X))
(PARTITION (VALUE (PORT IN1 (PART (NUM FA 4) FOO)) &X))
(PARTITION (VALUE (PORT IN2 (PART (NUM FA 1) FOO)) &X))
(PARTITION (VALUE (PORT IN2 (PART (NUM FA 2) FOO)) &X))
(PARTITION (VALUE (PORT IN2 (PART (NUM FA 3) FOO)) &X))
(PARTITION (VALUE (PORT IN2 (PART (NUM FA 4) FOO)) &X))
(PARTITION (VALUE (PORT CIN (PART (NUM FA 1) FOO)) &X))
(PARTITION (VALUE (PORT COUT (PART (NUM FA 1) FOO)) &X))
(PARTITION (VALUE (PORT COUT (PART (NUM FA 2) FOO)) &X))
(PARTITION (VALUE (PORT COUT (PART (NUM FA 3) FOO)) &X))
(PARTITION (VALUE (PORT COUT (PART (NUM FA 4) FOO)) &X))
(PARTITION (VALUE (PORT SUM (PART (NUM FA 1) FOO)) &X))
(PARTITION (VALUE (PORT SUM (PART (NUM FA 2) FOO)) &X))
```

```
(PARTITION (VALUE (PORT SUM (PART (NUM FA 3) FOO)) &X))
(PARTITION (VALUE (PORT SUM (PART (NUM FA 4) FOO)) &X))
(PARTITION (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT CIN (PART (NUM FA 2) FOO)) &X))
(PARTITION (VALUE (PORT CIN (PART (NUM FA 3) FOO)) &X))
(PARTITION (VALUE (PORT CIN (PART (NUM FA 4) FOO)) &X))
(PARTITION (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART AND2 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART AND1 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART OR1 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 1) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 1) FOO))) &X))
```

```
(PARTITION (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART AND2 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART AND1 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART OR1 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 2) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART AND2 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART AND1 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART OR1 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 3) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART AND2 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART AND1 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART OR1 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 4) FOO))) &X))
(PARTITION (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 4) FOO))) &X))
```

E.6 Multiprocessor Characteristics

E.6.1 Size of Multiprocessor

First, the number of processors used in the experiment was 61. This corresponds to an E-size of 5 (i.e., there are 5 processors on each side of the hexagonal surface). The

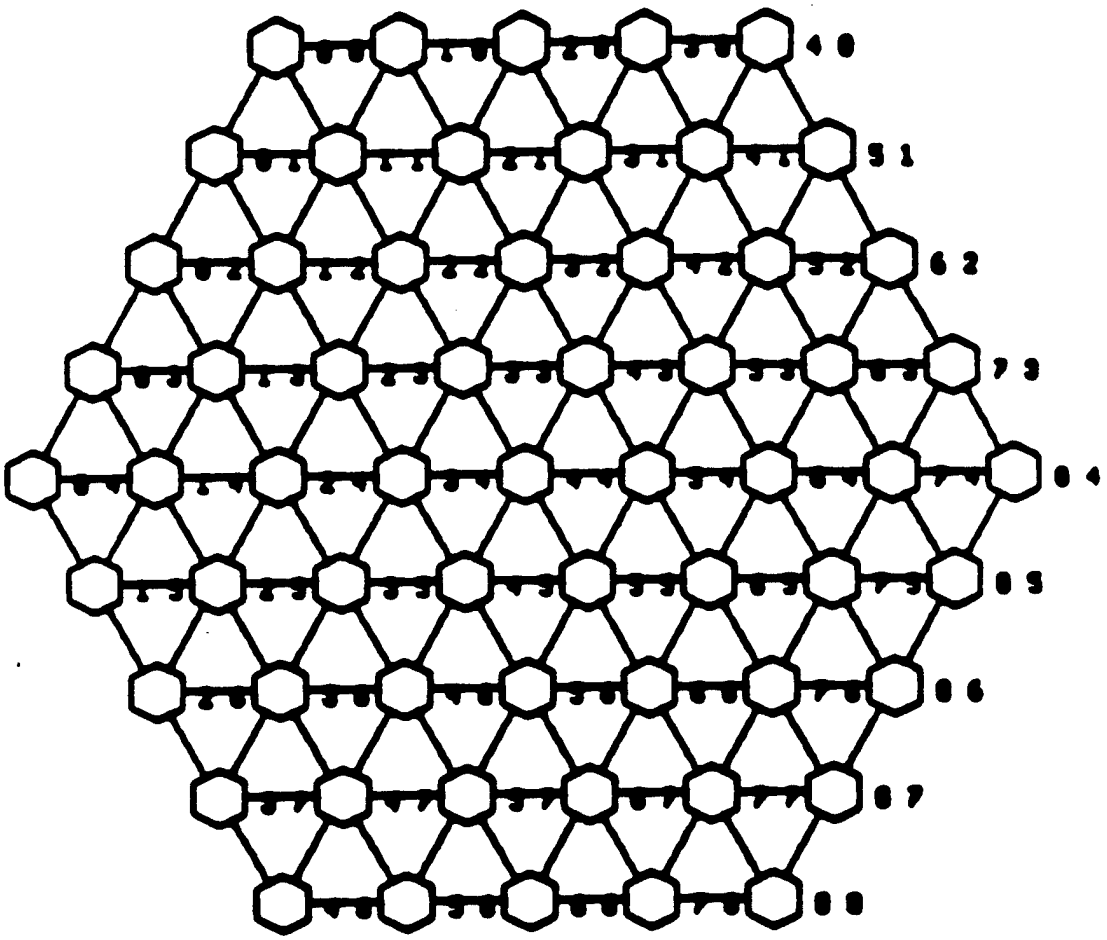


Figure 42: E-5 Processing Surface for FAIM-1

processors along with their processor addresses are shown in figure E.6.1. Wrap-around connections from the edges of the boundary are not shown in the figure for the sake of simplicity.

E.6.2 Processing Parameters

For the cost model described in chapter 3, the only constant given a non-zero value is K_U—the time taken to perform a successful unification. This constant is given the value 50 microseconds based on an estimate of 20 KLIPS for each processor.

E.6.3 Communication Parameters

The constants K_1,K_2, and K_3 used in the definition of the communication cost function (see chapter 3) have the values given below.

$K_1 = 2$ microseconds

$K_2 = 2$ microseconds/packet

$K_3 = 1$ microseconds/hop

All messages are assumed to fit in one packet.

E.7 Allocation Database

E.7.1 Allocation Database for Single Copy Case

In this database, facts of the form
(LOC <FACT-PATTERN> <PROCESSOR-ADDRESS>)
are intended to mean that the partition specified by
(PARTITION <FACT-PATTERN>)
should be allocated to the processor with the address <PROCESSOR-ADDRESS>.
The database is shown below.

```
(LOC (VALUE (PORT IN1 (PART (NUM FA 1.) FOO)) &X) (3. 7.))
(LOC (VALUE (PORT IN1 (PART (NUM FA 2.) FOO)) &X) (2. 4.))
(LOC (VALUE (PORT IN1 (PART (NUM FA 3.) FOO)) &X) (2. 6.))
(LOC (VALUE (PORT IN1 (PART (NUM FA 4.) FOO)) &X) (5. 4.))
(LOC (VALUE (PORT IN2 (PART (NUM FA 1.) FOO)) &X) (6. 2.))
(LOC (VALUE (PORT IN2 (PART (NUM FA 2.) FOO)) &X) (5. 6.))
(LOC (VALUE (PORT IN2 (PART (NUM FA 3.) FOO)) &X) (3. 2.))
(LOC (VALUE (PORT IN2 (PART (NUM FA 4.) FOO)) &X) (1. 2.))
(LOC (VALUE (PORT CIN (PART (NUM FA 1.) FOO)) &X) (3. 5.))
(LOC (VALUE (PORT COUT (PART (NUM FA 1.) FOO)) &X) (8. 8.))
(LOC (VALUE (PORT COUT (PART (NUM FA 2.) FOO)) &X) (6. 2.))
(LOC (VALUE (PORT COUT (PART (NUM FA 3.) FOO)) &X) (0. 0.))
(LOC (VALUE (PORT COUT (PART (NUM FA 4.) FOO)) &X) (4. 4.))
(LOC (VALUE (PORT SUM (PART (NUM FA 1.) FOO)) &X) NIL)
```

```
(LOC (VALUE (PORT SUM (PART (NUM FA 2.) FOO)) &X) NIL)
(LOC (VALUE (PORT SUM (PART (NUM FA 3.) FOO)) &X) NIL)
(LOC (VALUE (PORT SUM (PART (NUM FA 4.) FOO)) &X) NIL)
(LOC (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 1.) FOO))) &X) (3. 5.))
(LOC (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 2.) FOO))) &X) (6. 7.))
(LOC (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 3.) FOO))) &X) (4. 1.))
(LOC (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 4.) FOO))) &X) (7. 4.))
(LOC (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 1.) FOO))) &X) NIL)
(LOC (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 2.) FOO))) &X) NIL)
(LOC (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 3.) FOO))) &X) NIL)
(LOC (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 4.) FOO))) &X) NIL)
(LOC (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 1.) FOO))) &X) (6. 2.))
(LOC (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 2.) FOO))) &X) (4. 5.))
(LOC (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 3.) FOO))) &X) (3. 1.))
(LOC (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 4.) FOO))) &X) (4. 4.))
(LOC (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 1.) FOO))) &X) (6. 2.))
(LOC (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 2.) FOO))) &X) (7. 6.))
(LOC (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 3.) FOO))) &X) (5. 1.))
(LOC (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 4.) FOO))) &X) (0. 2.))
(LOC (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 1.) FOO))) &X) (2. 6.))
(LOC (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 2.) FOO))) &X) (3. 5.))
(LOC (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 3.) FOO))) &X) (4. 2.))
(LOC (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 4.) FOO))) &X) (5. 4.))
(LOC (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 1.) FOO))) &X) (3. 6.))
(LOC (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 2.) FOO))) &X) (5. 5.))
(LOC (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 3.) FOO))) &X) (2. 5.))
(LOC (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 4.) FOO))) &X) (0. 3.))
(LOC (VALUE (PORT CIN (PART (NUM FA 2.) FOO)) &X) (7. 7.))
(LOC (VALUE (PORT CIN (PART (NUM FA 3.) FOO)) &X) (5. 2.))
(LOC (VALUE (PORT CIN (PART (NUM FA 4.) FOO)) &X) (8. 4.))
(LOC (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 1.) FOO))) &X) (1. 5.))
(LOC (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 1.) FOO))) &X) (4. 0.))
(LOC (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 1.) FOO))) &X) (2. 5.))
(LOC (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 1.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART AND2 (PART (NUM FA 1.) FOO))) &X) (2. 5.))
(LOC (VALUE (PORT OUT (PART AND1 (PART (NUM FA 1.) FOO))) &X) (5. 1.))
(LOC (VALUE (PORT OUT (PART OR1 (PART (NUM FA 1.) FOO))) &X) (4. 0.))
(LOC (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 1.) FOO))) &X) NIL)
```

```
(LOC (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 1.) FOO))) &X) (3. 6.))
(LOC (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 2.) FOO))) &X) (4. 7.))
(LOC (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 2.) FOO))) &X) (3. 6.))
(LOC (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 2.) FOO))) &X) (5. 6.))
(LOC (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 2.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART AND2 (PART (NUM FA 2.) FOO))) &X) (5. 7.))
(LOC (VALUE (PORT OUT (PART AND1 (PART (NUM FA 2.) FOO))) &X) (4. 6.))
(LOC (VALUE (PORT OUT (PART OR1 (PART (NUM FA 2.) FOO))) &X) (3. 7.))
(LOC (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 2.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 2.) FOO))) &X) (6. 6.))
(LOC (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 3.) FOO))) &X) (2. 0.))
(LOC (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 3.) FOO))) &X) (2. 1.))
(LOC (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 3.) FOO))) &X) (4. 0.))
(LOC (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 3.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART AND2 (PART (NUM FA 3.) FOO))) &X) (3. 0.))
(LOC (VALUE (PORT OUT (PART AND1 (PART (NUM FA 3.) FOO))) &X) (3. 2.))
(LOC (VALUE (PORT OUT (PART OR1 (PART (NUM FA 3.) FOO))) &X) (1. 0.))
(LOC (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 3.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 3.) FOO))) &X) (1. 5.))
(LOC (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 4.) FOO))) &X) (5. 4.))
(LOC (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 4.) FOO))) &X) (4. 4.))
(LOC (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 4.) FOO))) &X) (7. 5.))
(LOC (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 4.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART AND2 (PART (NUM FA 4.) FOO))) &X) (6. 4.))
(LOC (VALUE (PORT OUT (PART AND1 (PART (NUM FA 4.) FOO))) &X) (4. 4.))
(LOC (VALUE (PORT OUT (PART OR1 (PART (NUM FA 4.) FOO))) &X) (4. 4.))
(LOC (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 4.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 4.) FOO))) &X) (8. 6.))
```

E.7.2 Allocation Database for Multiple Copy Case

In this database, facts of the form
(LOC <FACT-PATTERN> <CLUSTER>)
are intended to mean that the partition specified by
(PARTITION <FACT-PATTERN>)
should be allocated to the cluster of processors <CLUSTER>. A cluster specified

as (<n> *PA1* (<PA1> <PA2>...<PAn>) denotes <n> processors with the addresses <PA1>, <PA2>,...,<PAn>. *PA1* is the central processor in this cluster of processors. A cluster specified by *nil* indicates that no processors belong to the cluster (i.e., the related partition is not allocated to any processor). This is reasonable if the partition is not used at all for proving the goals specified at compile-time. The location database is given below.

```
(LOC (VALUE (PORT IN1 (PART (NUM FA 1.) FOO)) &X) (19. (4. 1.) ((1. 5.) (0. 4.) (8. 8.) (7. 8.) (2. 0.)
(2. 1.) (3. 2.) (4. 3.) (5. 3.) (6. 3.) (6. 2.) (2. 6.) (4. 0.) (3. 0.) (3. 1.) (4. 2.) (5. 2.) (5. 1.)
(4. 1.))))
(LOC (VALUE (PORT IN1 (PART (NUM FA 2.) FOO)) &X) (7. (2. 3.) ((2. 2.) (1. 2.) (1. 3.) (2. 4.) (3. 4.) (3.
3.) (2. 3.))))
(LOC (VALUE (PORT IN1 (PART (NUM FA 3.) FOO)) &X) (7. (2. 4.) ((2. 3.) (1. 3.) (1. 4.) (2. 5.) (3. 5.) (3.
4.) (2. 4.))))
(LOC (VALUE (PORT IN1 (PART (NUM FA 4.) FOO)) &X) (7. (4. 3.) ((4. 2.) (3. 2.) (3. 3.) (4. 4.) (5. 4.) (5.
3.) (4. 3.))))
(LOC (VALUE (PORT IN2 (PART (NUM FA 1.) FOO)) &X) (19. (0. 0.) ((6. 8.) (5. 7.) (4. 7.) (3. 7.) (7. 3.)
(7. 4.) (8. 5.) (0. 2.) (1. 2.) (2. 2.) (2. 1.) (2. 0.) (5. 8.) (4. 8.) (8. 4.) (0. 1.) (1. 1.) (1. 0.)
(0. 0.))))
(LOC (VALUE (PORT IN2 (PART (NUM FA 2.) FOO)) &X) (7. (4. 2.) ((4. 1.) (3. 1.) (3. 2.) (4. 3.) (5. 3.) (5.
2.) (4. 2.))))
(LOC (VALUE (PORT IN2 (PART (NUM FA 3.) FOO)) &X) (7. (7. 7.) ((7. 6.) (6. 6.) (6. 7.) (7. 8.) (8. 8.) (8.
7.) (7. 7.))))
(LOC (VALUE (PORT IN2 (PART (NUM FA 4.) FOO)) &X) (7. (3. 5.) ((3. 4.) (2. 4.) (2. 5.) (3. 6.) (4. 6.) (4.
5.) (3. 5.))))
(LOC (VALUE (PORT CIN (PART (NUM FA 1.) FOO)) &X) (19. (8. 4.) ((5. 8.) (4. 7.) (3. 7.) (6. 2.) (6. 3.)
(6. 4.) (7. 5.) (8. 6.) (0. 2.) (1. 2.) (1. 1.) (1. 0.) (4. 8.) (7. 3.) (7. 4.) (8. 5.) (0. 1.) (0. 0.)
(8. 4.))))
(LOC (VALUE (PORT COUT (PART (NUM FA 1.) FOO)) &X) (7. (7. 7.) ((7. 6.) (6. 6.) (6. 7.) (7. 8.) (8. 8.)
(8. 7.) (7. 7.))))
(LOC (VALUE (PORT COUT (PART (NUM FA 2.) FOO)) &X) (7. (5. 4.) ((5. 3.) (4. 3.) (4. 4.) (5. 5.) (6. 5.)
(6. 4.) (5. 4.))))
(LOC (VALUE (PORT COUT (PART (NUM FA 3.) FOO)) &X) (1. (5. 5.) ((5. 5.))))
(LOC (VALUE (PORT COUT (PART (NUM FA 4.) FOO)) &X) (1. (4. 4.) ((4. 4.))))
(LOC (VALUE (PORT SUM (PART (NUM FA 1.) FOO)) &X) NIL)
(LOC (VALUE (PORT SUM (PART (NUM FA 2.) FOO)) &X) NIL)
(LOC (VALUE (PORT SUM (PART (NUM FA 3.) FOO)) &X) NIL)
```

```
(LOC (VALUE (PORT SUM (PART (NUM FA 4.) FOO)) &X) NIL)
(LOC (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 1.) FOO))) &X) (19. (8. 4.) ((5. 8.) (4. 7.) (3. 7.) (6.
2.) (6. 3.) (6. 4.) (7. 5.) (8. 6.) (0. 2.) (1. 2.) (1. 1.) (1. 0.) (4. 8.) (7. 3.) (7. 4.) (8. 5.) (0.
1.) (0. 0.) (8. 4.))))
(LOC (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 2.) FOO))) &X) (7. (5. 5.) ((5. 4.) (4. 4.) (4. 5.) (5. 6.)
(6. 6.) (6. 5.) (5. 5.))))
(LOC (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 3.) FOO))) &X) (7. (2. 4.) ((2. 3.) (1. 3.) (1. 4.) (2. 5.)
(3. 5.) (3. 4.) (2. 4.))))
(LOC (VALUE (PORT IN1 (PART AND2 (PART (NUM FA 4.) FOO))) &X) (1. (4. 5.) ((4. 5.))))
(LOC (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 1.) FOO))) &X) NIL)
(LOC (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 2.) FOO))) &X) NIL)
(LOC (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 3.) FOO))) &X) NIL)
(LOC (VALUE (PORT IN2 (PART XOR2 (PART (NUM FA 4.) FOO))) &X) NIL)
(LOC (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 1.) FOO))) &X) (7. (0. 0.) ((5. 8.) (4. 8.) (8. 4.) (0. 1.)
(1. 1.) (1. 0.) (0. 0.))))
(LOC (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 2.) FOO))) &X) (7. (4. 2.) ((4. 1.) (3. 1.) (3. 2.) (4. 3.)
(5. 3.) (5. 2.) (4. 2.))))
(LOC (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 3.) FOO))) &X) (1. (3. 3.) ((3. 3.))))
(LOC (VALUE (PORT IN2 (PART AND1 (PART (NUM FA 4.) FOO))) &X) (1. (5. 4.) ((5. 4.))))
(LOC (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 1.) FOO))) &X) (19. (4. 5.) ((5. 4.) (4. 3.) (3. 3.) (2.
3.) (2. 4.) (2. 5.) (3. 6.) (4. 7.) (5. 7.) (6. 7.) (6. 6.) (6. 5.) (4. 4.) (3. 4.) (3. 5.) (4. 6.) (5.
6.) (5. 5.) (4. 5.))))
(LOC (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 2.) FOO))) &X) (7. (1. 5.) ((1. 4.) (0. 4.) (4. 0.) (5. 1.)
(2. 6.) (2. 5.) (1. 5.))))
(LOC (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 3.) FOO))) &X) (7. (7. 7.) ((7. 6.) (6. 6.) (6. 7.) (7. 8.)
(8. 8.) (8. 7.) (7. 7.))))
(LOC (VALUE (PORT IN2 (PART XOR1 (PART (NUM FA 4.) FOO))) &X) (1. (3. 5.) ((3. 5.))))
(LOC (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 1.) FOO))) &X) (19. (4. 4.) ((5. 3.) (4. 2.) (3. 2.) (2.
2.) (2. 3.) (2. 4.) (3. 5.) (4. 6.) (5. 6.) (6. 6.) (6. 5.) (6. 4.) (4. 3.) (3. 3.) (3. 4.) (4. 5.) (5.
5.) (5. 4.) (4. 4.))))
(LOC (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 2.) FOO))) &X) (7. (2. 3.) ((2. 2.) (1. 2.) (1. 3.) (2. 4.)
(3. 4.) (3. 3.) (2. 3.))))
(LOC (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 3.) FOO))) &X) (7. (2. 4.) ((2. 3.) (1. 3.) (1. 4.) (2. 5.)
(3. 5.) (3. 4.) (2. 4.))))
(LOC (VALUE (PORT IN1 (PART AND1 (PART (NUM FA 4.) FOO))) &X) (1. (4. 3.) ((4. 3.))))
(LOC (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 1.) FOO))) &X) (19. (4. 1.) ((1. 5.) (0. 4.) (8. 8.) (7.
8.) (2. 0.) (2. 1.) (3. 2.) (4. 3.) (5. 3.) (6. 3.) (6. 2.) (2. 6.) (4. 0.) (3. 0.) (3. 1.) (4. 2.) (5.
2.) (5. 1.) (4. 1.))))
```

```
(LOC (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 2.) FOO))) &X) (7. (7. 4.) ((7. 3.) (6. 3.) (6. 4.) (7. 5.)
(8. 5.) (8. 4.) (7. 4.))))
(LOC (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 3.) FOO))) &X) (7. (4. 6.) ((4. 5.) (3. 5.) (3. 6.) (4. 7.)
(5. 7.) (5. 6.) (4. 6.))))
(LOC (VALUE (PORT IN1 (PART XOR1 (PART (NUM FA 4.) FOO))) &X) (1. (2. 3.) ((2. 3.))))
(LOC (VALUE (PORT CIN (PART (NUM FA 2.) FOO)) &X) (7. (8. 7.) ((8. 6.) (7. 6.) (7. 7.) (8. 8.) (0. 4.) (0.
3.) (8. 7.))))
(LOC (VALUE (PORT CIN (PART (NUM FA 3.) FOO)) &X) (7. (2. 4.) ((2. 3.) (1. 3.) (1. 4.) (2. 5.) (3. 5.) (3.
4.) (2. 4.))))
(LOC (VALUE (PORT CIN (PART (NUM FA 4.) FOO)) &X) (1. (5. 6.) ((5. 6.))))
(LOC (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 1.) FOO))) &X) (19. (4. 5.) ((5. 4.) (4. 3.) (3. 3.) (2. 3.)
(2. 4.) (2. 5.) (3. 6.) (4. 7.) (5. 7.) (6. 7.) (6. 6.) (6. 5.) (4. 4.) (3. 4.) (3. 5.) (4. 6.) (5. 6.)
(5. 5.) (4. 5.))))
(LOC (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 1.) FOO))) &X) (19. (0. 0.) ((6. 8.) (5. 7.) (4. 7.) (3. 7.)
(7. 3.) (7. 4.) (8. 5.) (0. 2.) (1. 2.) (2. 2.) (2. 1.) (2. 0.) (5. 8.) (4. 8.) (8. 4.) (0. 1.) (1. 1.)
(1. 0.) (0. 0.))))
(LOC (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 1.) FOO))) &X) (19. (4. 0.) ((1. 4.) (0. 3.) (8. 7.) (7.
7.) (7. 8.) (2. 0.) (3. 1.) (4. 2.) (5. 2.) (6. 2.) (2. 6.) (2. 5.) (0. 4.) (8. 8.) (3. 0.) (4. 1.) (5.
1.) (1. 5.) (4. 0.))))
(LOC (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 1.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART AND2 (PART (NUM FA 1.) FOO))) &X) (19. (4. 0.) ((1. 4.) (0. 3.) (8. 7.) (7.
7.) (7. 8.) (2. 0.) (3. 1.) (4. 2.) (5. 2.) (6. 2.) (2. 6.) (2. 5.) (0. 4.) (8. 8.) (3. 0.) (4. 1.) (5.
1.) (1. 5.) (4. 0.))))
(LOC (VALUE (PORT OUT (PART AND1 (PART (NUM FA 1.) FOO))) &X) (19. (0. 0.) ((6. 8.) (5. 7.) (4. 7.) (3.
7.) (7. 3.) (7. 4.) (8. 5.) (0. 2.) (1. 2.) (2. 2.) (2. 1.) (2. 0.) (5. 8.) (4. 8.) (8. 4.) (0. 1.) (1.
1.) (1. 0.) (0. 0.))))
(LOC (VALUE (PORT OUT (PART OR1 (PART (NUM FA 1.) FOO))) &X) (19. (4. 5.) ((5. 4.) (4. 3.) (3. 3.) (2. 3.)
(2. 4.) (2. 5.) (3. 6.) (4. 7.) (5. 7.) (6. 7.) (6. 6.) (6. 5.) (4. 4.) (3. 4.) (3. 5.) (4. 6.) (5. 6.)
(5. 5.) (4. 5.))))
(LOC (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 1.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 1.) FOO))) &X) (19. (4. 5.) ((5. 4.) (4. 3.) (3. 3.) (2.
3.) (2. 4.) (2. 5.) (3. 6.) (4. 7.) (5. 7.) (6. 7.) (6. 6.) (6. 5.) (4. 4.) (3. 4.) (3. 5.) (4. 6.) (5.
6.) (5. 5.) (4. 5.))))
(LOC (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 2.) FOO))) &X) (7. (4. 2.) ((4. 1.) (3. 1.) (3. 2.) (4. 3.)
(5. 3.) (5. 2.) (4. 2.))))
(LOC (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 2.) FOO))) &X) (7. (2. 3.) ((2. 2.) (1. 2.) (1. 3.) (2. 4.)
(3. 4.) (3. 3.) (2. 3.))))
```

```
(LOC (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 2.) FOO))) &X) (7. (4. 7.) ((4. 6.) (3. 6.) (3. 7.) (4. 8.)
(5. 8.) (5. 7.) (4. 7.))))
(LOC (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 2.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART AND2 (PART (NUM FA 2.) FOO))) &X) (7. (7. 4.) ((7. 3.) (6. 3.) (6. 4.) (7. 5.)
(8. 5.) (8. 4.) (7. 4.))))
(LOC (VALUE (PORT OUT (PART AND1 (PART (NUM FA 2.) FOO))) &X) (7. (2. 3.) ((2. 2.) (1. 2.) (1. 3.) (2. 4.)
(3. 4.) (3. 3.) (2. 3.))))
(LOC (VALUE (PORT OUT (PART OR1 (PART (NUM FA 2.) FOO))) &X) (7. (4. 2.) ((4. 1.) (3. 1.) (3. 2.) (4. 3.)
(5. 3.) (5. 2.) (4. 2.))))
(LOC (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 2.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 2.) FOO))) &X) (7. (1. 0.) ((6. 8.) (5. 8.) (0. 0.) (1. 1.)
(2. 1.) (2. 0.) (1. 0.))))
(LOC (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 3.) FOO))) &X) (7. (5. 5.) ((5. 4.) (4. 4.) (4. 5.) (5. 6.)
(6. 6.) (6. 5.) (5. 5.))))
(LOC (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 3.) FOO))) &X) (7. (5. 5.) ((5. 4.) (4. 4.) (4. 5.) (5. 6.)
(6. 6.) (6. 5.) (5. 5.))))
(LOC (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 3.) FOO))) &X) (7. (5. 5.) ((5. 4.) (4. 4.) (4. 5.) (5. 6.)
(6. 6.) (6. 5.) (5. 5.))))
(LOC (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 3.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART AND2 (PART (NUM FA 3.) FOO))) &X) (7. (5. 5.) ((5. 4.) (4. 4.) (4. 5.) (5. 6.)
(6. 6.) (6. 5.) (5. 5.))))
(LOC (VALUE (PORT OUT (PART AND1 (PART (NUM FA 3.) FOO))) &X) (7. (5. 5.) ((5. 4.) (4. 4.) (4. 5.) (5. 6.)
(6. 6.) (6. 5.) (5. 5.))))
(LOC (VALUE (PORT OUT (PART OR1 (PART (NUM FA 3.) FOO))) &X) (7. (5. 5.) ((5. 4.) (4. 4.) (4. 5.) (5. 6.)
(6. 6.) (6. 5.) (5. 5.))))
(LOC (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 3.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 3.) FOO))) &X) (7. (5. 5.) ((5. 4.) (4. 4.) (4. 5.) (5. 6.)
(6. 6.) (6. 5.) (5. 5.))))
(LOC (VALUE (PORT IN1 (PART OR1 (PART (NUM FA 4.) FOO))) &X) (1. (5. 4.) ((5. 4.))))
(LOC (VALUE (PORT IN2 (PART OR1 (PART (NUM FA 4.) FOO))) &X) (1. (4. 4.) ((4. 4.))))
(LOC (VALUE (PORT IN2 (PART AND2 (PART (NUM FA 4.) FOO))) &X) (1. (4. 4.) ((4. 4.))))
(LOC (VALUE (PORT IN1 (PART XOR2 (PART (NUM FA 4.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART AND2 (PART (NUM FA 4.) FOO))) &X) (1. (5. 5.) ((5. 5.))))
(LOC (VALUE (PORT OUT (PART AND1 (PART (NUM FA 4.) FOO))) &X) (1. (5. 4.) ((5. 4.))))
(LOC (VALUE (PORT OUT (PART OR1 (PART (NUM FA 4.) FOO))) &X) (1. (4. 4.) ((4. 4.))))
(LOC (VALUE (PORT OUT (PART XOR2 (PART (NUM FA 4.) FOO))) &X) NIL)
(LOC (VALUE (PORT OUT (PART XOR1 (PART (NUM FA 4.) FOO))) &X) (1. (3. 4.) ((3. 4.))))
```

Bibliography

[1] G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* Technical Report 844, MIT AI Laboratory, March 1985.

[2] A. V. Aho, M. R. Garey, and J. D. Ullman. The Transitive Reduction of a Directed Graph. *SIAM Journal on Computing*, 1(2):131–137, June 1972.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, 1983.

[4] Bill Athas. *Fine Grain Concurrent Computations.* PhD thesis, Computer Science Department, California Institute of Technology, 1987. Also published as technical report 5242:TR:87.

[5] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[6] A. Barr and E. A. Feigenbaum, editors. *The Handbook of Artificial Intelligence*, chapter 12, page 80. Volume 3, William Kauffman, Inc., Los Altos, California, 1982.

[7] A. Barr and E. A. Feigenbaum, editors. *The Handbook of Artificial Intelligence*, chapter 2, page 39. Volume 1, William Kauffman, Inc., Los Altos, California, 1981.

[8] L. Bic. A Data-Driven Model for Parallel Interpretation of Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 517–523, ICOT, 1984.

[9] P. Borgwardt. Parallel Prolog using Stack Segments on Shared-Memory Multiprocessors. In *IEEE Logic Programming Conference*, pages 2–11, IEEE, February 1984.

[10] Michael L. Campbell. Static Allocation for a Data Flow Multiprocessor. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 511–517, IEEE, 1985.

[11] Martha J. Chamberlain and A.L. Davis. A Static Resource Allocation Methodology for a Dataflow Multiprocessor. Copies may be obtained from A.L. Davis at Hewlett Packard Labs, 1501 Page Mill Rd., Bldg. 3U, Palo Alto, CA 94304.

[12] A. Ciepielewski and S. Haridi. A Formal Model for Or-Parallel Execution of Logic Programs. In *Proceedings of the IFIP Congress*, pages 299–305, IFIP, 1983.

[13] Wayne Citrin. Parallel Unification Scheduling in Prolog. 1985. Can be obtained from the Aquarius group, 512 Evans Hall, Berkeley, CA 94720.

[14] K. L. Clark and S. Gregory. A Relational Language for Parallel Programming. In *Proceedings of the Conference on Funcional Programming and Computer Architecture*, pages 171–178, Association for Computing Machinery, October 1981.

[15] John S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, University of California, Irvine, 1983.

[16] John S. Conery and Dennis F. Kibler. AND Parallelism in Logic Programs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 539–543, 1983.

[17] John S. Conery and Dennis F. Kibler. Parallel Interpretation of Logic Programs. In *Proceedings of the Conference on Functional Programming and Computer Architecture*, pages 163–170, Association for Computing Machinery, October 1981.

[18] A. L. Davis and S. V. Robison. The Architecture of the FAIM-1 Symbolic Multiprocessing System. In *Proceedings of IJCAI-85*, Morgan Kaufmann Publishers, Inc., August 1985.

[19] D. DeGroot. Restricted And-Parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 471–478, ICOT, Japan, 1984.

[20] C. Dwork, P. C. Kanellakis, and J. C. Mitchell. On the Sequential Nature of Unification. *The Journal of Logic Programming*, 1(1):35–50, June 1984.

[21] J. Finger and M. Genesereth. *Residue: A Deductive Approach to Design Synthesis.* Technical Report HPP-85-1, Heuristic Programming Project, Computer Science Department, Stanford University, January 1985.

[22] M. J. Flynn. Very High-Speed Computing Systems. *Proceedings of IEEE*, 54:1901–1909, December 1966.

[23] Charles L. Forgy. *OPS5 User's Manual.* Technical Report CMU-CS-81-135, Computer Science Department, Carnegie Mellon University, 1981.

[24] Gordon Foyster. *Helios: User's Manual.* Technical Report HPP-84-34, Heuristic Programming Project, Computer Science Department, Stanford University, August 1984.

[25] K. Furukawa, K. Nitta, and Y. Matsumoto. Prolog Interpreter Based on Concurrent Programming. In *Proceedings of the First International Logic Programming Conference*, pages 38–44, 1982.

[26] R.P. Gabriel and J. McCarthy. Queue-based Multi-processing Lisp. In *1984 ACM Symposium on Lisp and Functional Programming*, pages 25–44, August 1984.

[27] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, 1979.

[28] M. R. Genesereth. *The Use of Design Descriptions in Automated Diagnosis.* Technical Report HPP-81-20, Heuristic Programming Project, Computer Science Department, Stanford University, 1984.

[29] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, March 1969.

[30] S. Gregory. *Design, Application and Implementation of a Parallel Logic Programming Language.* PhD thesis, Imperial College of Science and Technology, 1985.

[31] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[32] R. Halstead. Parallel symbolic computing. *IEEE Computer*, 19(8):35–43, August 1986.

[33] Haridi, Seif and Ciepielewski, Andrzej. *An Or-Parallel Token Machine.* Technical Report TRITA-CS-8303, Department of Telecommunication Systems - Computer Systems, The Royal Institute of Technology, Sweden, May 1983.

[34] M.V. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-parallel Execution of Logic Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, 1987.

[35] Hillis, W. Daniel. *The Connection Machine.* A.I. Memo 646, Artificial Intelligence Laboratory, M.I.T., September 1981. Revised report under preparation.

[36] D.A. Hornig. *Automatic Partitioning and Scheduling on a Network of Personal Computers.* PhD thesis, Department of Computer Science, Carnegie Mellon University, 1984. Also available as technical report CMU-CS-84-165.

[37] B. J. Lageweg, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. *Computer Aided Complexity Classification of Deterministic Scheduling Problems.* Technical Report BW 138/81, Mathematisch Centrum, Amsterdam, 1981.

[38] B. W. Lampson, editor. *Distributed Systems. Lecture Notes in Computer Science*, Springer-Verlag, 1981.

[39] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinooy Kan. *Recent Developments in Deterministic Scheduling.* Technical Report BW 146/81, Mathematisch Centrum, Amsterdam, 1982.

[40] P.P. Li. *A Parallel Execution Model for Logic Programming.* PhD thesis, Computer Science Department, California Institute of Technology, 1986. Also published as technical report 5227:TR:86.

[41] G. Lindstrom and P. Panangaden. Stream-Based Execution of Logic Programs. In *IEEE Logic Programming Conference*, pages 168–176, IEEE, February 1984.

[42] Malone, T. W., R. E. Fikes, and M. T. Howard. *Enterprise: A Market-like Task Scheduler for Distributed Computing Environments.* Working Paper, Cognitive and Instructional Sciences Group, Xerox Palo Alto Research Center, Palo Alto, California, October 1983.

[43] E. W. Mayr. *Well Structured Parallel Programs Are Not Easier to Schedule.* Report No. STAN-CS-81-880, Stanford University, September 1981.

[44] Moon, David A. Architecture of the Symbolics 3600. In *The Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 76–83, 1985.

[45] T. Moto-oka and H. S. Stone. Fifth Generation Computer Systems: A Japanese Project. *Computer*, 17(3):6–13, March 1984.

[46] *Multimax Technical Summary.* Encore Computer Corporation, 257 Cedar Hill Street, Marlborough, MA 01752.

[47] B.J. Nelson. *Remote Procedure Call.* PhD thesis, Department of Computer Science, Carnegie Mellon University, 1981.

[48] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company, 1980.

[49] Kemal Oflazer. Partitioning in Parallel Processing of Production Systems. In *Proceedings of the International Conference on Parallel Processing*, IEEE, 1984.

[50] Kemal Oflazer. *Partitioning in Parallel Processing of Production Systems.* PhD thesis, Carnegie Mellon University, March 1987.

[51] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc, 1982.

[52] S. Pappert. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, 1980.

[53] Ian Robinson. A Prolog Processor Based on a Pattern Matching Memory Device. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 172–179, Springer-Verlag, July 1986.

[54] Stuart Russell. *The Compleat Guide to MRS.* Technical Report KSL-85-12, Knowledge Systems Laboratory, Computer Science Department, Stanford University, June 1985.

[55] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* PhD thesis, Electrical Engineering Department, Stanford University, April 1987. Also available as Computer Systems Laboratory Technical Report No. CSL-TR-87-328.

[56] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, 1985.

[57] E. Y. Shapiro. *A Subset of Concurrent Prolog and Its Interpreter.* Technical Report TR-003, ICOT, Japan, January 1983.

[58] Ehud Shapiro. Systolic programming: a paradigm of parallel processing. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, 1984.

[59] N. Singh. *Exploiting Design Morphology to Manage Complexity.* PhD thesis, Electrical Engineering Department, Stanford University, August 1985.

[60] N. Singh. *MARS: A Multiple Abstraction Rule-Based Simulator.* Technical Report HPP-83-43, Heuristic Programming Project, Computer Science Department, Stanford University, 1983.

[61] Vineet Singh and Michael R. Genesereth. A Variable Supply Model for Distributing Deductions. In *Proceedings of IJCAI-85*, Morgan Kaufmann Publishers Inc., August 1985.

[62] Vineet Singh and Michael R. Genesereth. PM: A Parallel Execution Model for Backward-Chaining Deductions. *Future Computing Systems*, 1(3):271–308, 1986. Also available as KSL Report KSL-85-18, May 1985, Knowledge Systems Laboratory, Computer Science Department, Stanford University.

[63] Vineet Singh and Michael R. Genesereth. *PM: A Parallel Execution Model for Backward-Chaining Deductions.* KSL Report KSL-85-18, Knowledge Systems Laboratory, Computer Science Department, Stanford University, May 1985.

[64] D. E. Smith and M. R. Genesereth. Ordering Conjunctive Queries. *Artificial Intelligence*, 26(2):171–215, October 1985.

[65] Smith, D. E. *Controlling Inference.* PhD thesis, Department of Computer Science, Stanford University, August 1985.

[66] Smith, R. G. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, December 1980.

[67] K. S. Stevens. 1985. Private communication.

[68] K. S. Stevens, S. V. Robison, and A. L. Davis. The Post Office—Communication Support for Distributed Ensemble Architectures. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 160–166, IEEE Computer Society, May 1986.

[69] Richard Treitel. *Sequentialization of Logic Programs.* PhD thesis, Stanford University, September 1986. Also available as Report No. STAN-CS-86-1135, Department of Computer Science, Stanford University, Stanford, CA 94305.

[70] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins. Data-Driven and Demand-Driven Computer Architecture. *Computing Surveys*, 14(1):93–143, March 1982.

[71] K. Ueda. *Guarded Horn Clauses.* Technical Report TR-103, ICOT, Tokyo, 1985.

[72] D.H.D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, AI Center, Computer Science and Technology Division, 1983.

[73] D.H.D. Warren. *Implementing Prolog—Compiling Predicate Logic Programs.* Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[74] H. Yasuura. On Parallel Computational Complexity of Unification. In *Proceedings of International Conference on Fifth Generation Computers*, Tokyo, Japan, November 1984.

Reproduced by NTIS

National Technical Information Service
Springfield, VA 22161

This report was printed specifically for your order from nearly 3 million titles available in our collection.

For economy and efficiency, NTIS does not maintain stock of its vast collection of technical reports. Rather, most documents are printed for each order. Documents that are not in electronic format are reproduced from master archival copies and are the best possible reproductions available. If you have any questions concerning this document or any order you have placed with NTIS, please call our Customer Service Department at (703) 487-4660.

About NTIS

NTIS collects scientific, technical, engineering, and business related information — then organizes, maintains, and disseminates that information in a variety of formats — from microfiche to online services. The NTIS collection of nearly 3 million titles includes reports describing research conducted or sponsored by federal agencies and their contractors; statistical and business information; U.S. military publications; audiovisual products; computer software and electronic databases developed by federal agencies; training tools; and technical reports prepared by research organizations worldwide. Approximately 100,000 *new* titles are added and indexed into the NTIS collection annually.

For more information about NTIS products and services, call NTIS at (703) 487-4650 and request the free *NTIS Catalog of Products and Services*, PR-827LPG, or visit the NTIS Web site **http://www.ntis.gov**.

NTIS
Your indispensable resource for government-sponsored information—U.S. and worldwide